AFRL-RI-RS-TR-2019-015

TRUSTED AND RESILIENT MISSION OPERATION

RECTOR & VISITORS OF THE UNIVERSITY OF VIRGINIA

JANUARY 2019

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2019-015 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /	**/ S /**
WILMAR SIFRE	STEVEN JOHNS
Work Unit Manager	Chief, Trusted Systems Branch
	Computing & Communications Division
	Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* JANUARY 2019	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* JUN 2017 – SEP 2018

4. TITLE AND SUBTITLE TRUSTED AND RESILIENT MISSION OPERATION	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA8750-17-2-0079
	5c. PROGRAM ELEMENT NUMBER 63788F
6. AUTHOR(S) Jack Davidson	**5d. PROJECT NUMBER** T3ET
	5e. TASK NUMBER UN
	5f. WORK UNIT NUMBER VA
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rector & Visitors of the University of Virginia, 1001 N Emmet St, Charlottesville VA 22903-4833	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2019-015

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Cyber physical systems (CPS) form a ubiquitous, networked computing substrate, which is increasingly essential to our nation's civilian and military infrastructure. These systems must be highly resilient to adversaries, perform mission critical functions despite known/unknown vulnerabilities, and protect and repair themselves during or after operational failures and cyber-attacks. We believe that an automated CPS repair approach that can prevent failures of related, mission-critical systems is a necessary component to support the resiliency and survivability of our nation's infrastructure. We integrated and evaluated techniques to cooperatively eliminate certain security vulnerabilities in CPS, to repair certain general classes of such systems, and to increase the confidence of human operators in the trustworthiness of those repairs and the subsequent system behavior. We worked with a Government-provided Red Team to demonstrate and validate our approach on embedded platforms, including an autonomous rover vehicle.

15. SUBJECT TERMS
Cyber physical systems, autonomous vehicle, cyber security

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**WILMAR SIFRE**
U	U	U	UU	21	**19b. TELEPHONE NUMBER (*Include area code*)** **N/A**

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

1. Summary

Cyber physical systems (CPS) form a ubiquitous, networked computing substrate, which is increasingly essential to our nation's civilian and military infrastructure. These systems must be highly resilient to adversaries, perform mission critical functions despite known and unknown vulnerabilities, and protect and repair themselves during or after operational failures and cyber-attacks. We believe that an automated CPS repair approach that can prevent failures of related, mission-critical systems is a necessary component to support the resiliency and survivability of our nation's infrastructure. We integrated and evaluated previously-developed techniques to cooperatively eliminate certain software security vulnerabilities in cyber physical systems, to repair certain general classes of such systems, and to increase the confidence of human operators in the trustworthiness of those repairs and the subsequent system behavior. We worked with a Government-provided Red Team to demonstrate and validate our approach on embedded platforms, including an autonomous rover vehicle.

2. Introduction

The main thrusts of our work in Trusted and Resilient Mission Operation were:

- Integrated and evaluated software hardening techniques into an autonomous vehicle system
    - We worked with a Government-furnished Red Team to assess resiliency and overhead of our integration effort. We integrated our technologies both in software simulation and in a physical autonomous vehicle platform.
- Improved and targeted established best-of-breed techniques to the domain of autonomous vehicles.
    - We applied software transformation techniques to harden vehicle control software against attacks by increasing software diversity. These transformations were capable of both detecting and mitigating many classes of attacks.
    - We enhanced operator trust through runtime monitoring on autonomous vehicles. Leveraging lightweight techniques for measuring system behavior, we contributed to improved operator trust in our platform.
    - We adapted automatic program repair techniques to repair bugs in vehicle control software, with a focus on the types of bugs repaired and the quality of repairs.
    - We developed a method for measuring the quality of automatically-generated software repairs via dynamic invariant detection.
    - We developed formal proofs of architectural properties of trusted and resilient our autonomous vehicle system, including its defense-in-depth dual-controller design. We proved that a compromised vehicle retains trustworthy control of the physical platform.

3. Methods, Assumptions, and Procedures

Over the course of this project, we integrated multiple software hardening, diversity, repair, and monitoring techniques to improve trust and resilience in cyber physical systems (CPS). We implemented these techniques on a prototype autonomous vehicle consisting of commercially-available embedded devices (Intel UP boards) running open-source autonomous vehicle control software (ArduPilot) on a physical rover platform. These techniques operate on the source or binary code of a program (e.g., the ArduPilot software that controls a vehicle) and transform it to automatically mitigate malicious behavior, detect malicious behavior, fight through detected malicious or defective behavior, or otherwise enhance the trustworthiness and resilience of the original piece of software. Working in concert with a Red Team, we integrated these techniques into a single prototype autonomous vehicle platform to demonstrate the viability of broad deployment against indicative, realistic attacks.

3.1. Mission Support Platform

We developed an indicative prototype autonomous vehicle based on two Intel UP boards, each running the open source ArduPilot software package on a custom-built physical rover platform. We tested this vehicle using waypoint-based missions at the University of Michigan's M-AIR facility. This prototype vehicle uses off-the-shelf components and open-source software. We designed a built this prototype to use in tandem with our collaborators and with the Red Team.

At a high level, our prototype consists of two Intel UP boards. One board, the untrusted *locomotion controller*, runs a given mission (e.g., to move to waypoints or accomplish some task). Meanwhile, the other board, the *trusted controller*, runs our trust measurement tools to assess whether the vehicle is operating as expected. We refer to this as a *dual controller* design. If an attack occurs against the locomotion controller, the trusted controller takes physical control of the vehicle so that it can synthesize a repair to the software to eliminate, mitigate, or limit the attack. After synthesizing and deploying a repair, the locomotion controller regains control of the vehicle and resumes mission execution.

3.2. Red Team Evaluation Overview

We worked with an independent Red Team, who evaluated our prototype in two parts. First, we agreed on the prototype rover (as described in Section 3.1) as well as a mission for the rover to complete. The Red Team provided a total of fourteen defects in the ArduPilot codebase indicative of real-world security vulnerabilities against CPSs. The Red Team would deploy the attack while the rover was completing the mission—if successful, the attack would cause the rover to behave abnormally (e.g., to miss a waypoint or to crash the software). Ultimately, we sought to improve system resilience and trustworthiness by (1) hardening the software to eliminate the vulnerability altogether, (2) detect violations of trust stemming from the vulnerability, (3) if trust is violated, directing the rover to navigate to a safe-zone while synthesizing a repair, and (4) deploy the repair to the rover, at which point it can continue its mission unhindered.

During the first half of the project, the Red Team developed these defect scenarios and tested our integration effort in software simulation by measuring how many of the scenarios were detected or mitigated during execution of an indicative mission as well as the time, space, and networking overhead associated with the deployment of our techniques. The Red Team also received a physical rover prototype to evaluate a subset of these defects during the second half of this project, culminating in a live demonstration of techniques working in concert.

3.3. Software Hardening via Binary Rewriting ("Zipr")

Our prototype system first transforms an input program in a way that makes it more difficult to attack, increasing the overall trustworthiness of the system. We developed, integrated, and evaluated techniques for secure operation in our representative rover prototype. Through the application of our *Zipr* toolchain, we successfully applied a wide range of diversity and hardening transformations to CPS binary code without requiring the availability of source code or debugging symbols. We applied Zipr to the ArduPilot software running on our prototype's locomotion controller.

The main software hardening achievements during the project were:

- Demonstrating the applicability of Zipr algorithms to CPS software
- Demonstrating the effectiveness of composing multiple security and diversity transformations to prevent exploits
- Developing, integrating and evaluating the ability to rewrite and compose security transformations for binaries containing C++ exception tables
- Refining reverse engineering techniques to increase the precision of security transformations, resulting in reduced attack surfaces
- Improving the performance of our control-flow integrity implementation
- Paving the path for techniques to not only detect, but also tolerate, attacks, without loss of service

Zipr can be applied to a system before that system is deployed. Zipr enhances the overall trustworthiness of a system by reducing opportunities for an attacker to compromise the system. The Zipr effort was led by the University of Virginia team.

3.4. Runtime Monitoring via Composable and Measurable Views of Trust

Once Zipr has been applied to transform an input program, we next use a combination of statistical techniques and human operator assessment to (1) model the normal operation of the system, and (2) monitor for anomalies that deviate from this normal operation. Our runtime monitoring and verification technology builds upon recent project work under the AFRL-funded Composable and Measurable view of Trust (CMT) project. During this project, CMT techniques and tools for modeling and monitoring trust were enhanced to support x86 ArduPilot platforms and project specific mission scenarios. A defining aspect of this effort was to understand and respond to the requirements and interplay of the different technologies, while ensuring that the concert of technologies cooperate with each other without interference. The main trust assessment and runtime monitoring contributions in this project include:

- Enabling continuous trust monitoring of autonomous vehicle mission operation
- Reporting trust violations and observable preconditions leading to violations
- Visualizing mission progress, attack locations and trust criteria for demonstrations
- Supporting the design, implementation and fielding of a team-wide integrated software/hardware system.

CMT increases trust through the use of runtime monitoring to ensure the system behaves according to a model of correct behavior provided by a human operator. The CMT effort was led by the Raytheon/BBN team.

3.5. Automated Software Repair via Darjeeling

If CMT or Zipr indicate that a trust violation has occurred, we use automated program repair to synthesize a new version of the software that enables resiliency by providing immunity to the vulnerability leading to that violation. We developed an automated program repair tool, called *Darjeeling*, targeting the Intel UP boards used in our prototype rover. Darjeeling is a modern implementation of the GenProg family of repair algorithms, evaluated in previous DARPA and Air Force efforts. Darjeeling is capable of repairing many autonomous vehicle software vulnerabilities. In our prototype, the trusted controller uses Darjeeling to produce a repaired version of the ArduPilot software, which is then deployed on the locomotion controller. This allows the locomotion controller to resume mission operation without subsequently succumbing to the original vulnerability.

The repair process is split into an offline and an online stage:

The offline stage of repair involves the integration of several off-the-shelf software components:

- We adapt each of the defect scenarios provided by the Red Team to work within a *containerization* platform for reliably reproducing and analyzing buggy behavior. Containerization works by building an isolated computing environment comprised of the source code, binary software, and libraries for the ArduPilot system used in our prototype.

- Darjeeling computes a set of static analyses on the program and builds a database of donor code snippets which supply the code used to craft repairs. Darjeeling uses the results of these static analyses during repair to identify and prune redundant program transformations.

The offline stage served as an optimization during the Red Team evaluation. We note that, in practice, an attacker would not make available the source code of a vulnerability. Instead, the same steps could be computed in the online stage based upon information provided by Zipr (Section 3.3) or CMT (Section 3.4).

The online stage of the repair process begins when the trusted controller supplies the repair module with details of an attack, allowing the attack to be reproduced in simulation. Critically, this step occurs on the trusted controller of the rover—we can successfully synthesize program repairs entirely in the self-contained rover platform without the use of external computing (e.g., cloud) resources. This online stage continues until Darjeeling provides one or more acceptable patches to the trusted controller, or else exhausts its allocated resources:

- The repair module safely recreates the attack in simulation using an instrumented binary inside a sandboxed container to identify source code regions associated with the attack. This information is combined with the offline-computed information to determine a set of candidate repair locations.
- Darjeeling combines the set of candidate repair locations with the precomputed static analysis, code snippet database, and a set of repair templates to generate the space of concrete program transformations.
- Darjeeling exhaustively searches through the set of program transformations for an acceptable repair. To determine whether a candidate patch is acceptable, Darjeeling evaluates its corresponding binary on a test suite of missions, including one that reproduces the attack. An acceptable patch is one that addresses the vulnerability while retaining existing functionality (i.e., the patched program passes all tests).

We make several choices to support efficient repair search in the resource-constrained environment of the rover. Most importantly, Darjeeling evaluates candidate patches using a low fidelity simulation in containers. This allow multiple patches to be evaluated in parallel because it incurs minimal resource overhead. Additionally, the flow of time can be accelerated in simulation, permitting more repairs to be found within a fixed window of time.

Darjeeling provides system resiliency by transforming software in a way that grants immunity to observed anomalous behavior. The repair effort was led by the Carnegie Mellon team.

3.6. Invariant Analysis

As automated repair techniques may produce candidate repairs to software that are potentially undesirable, we may favor trusting a repair that retains original functionality over a repair that removes significant amounts of functionality. Thus, we adapted invariant analysis techniques to our integration effort to assess the similarity of patches synthesized by Darjeeling (Section 4.4). At a high level, this takes the form of a two-stage process. First, invariants are inferred from the static source code and runtime behavior of a given system. This inference is applied to the original buggy code as well as to generated candidate patches. Second, the invariant sets associated with various patches and the original program are compared for similarity. A similarity metric allows us to more rapidly assess the acceptability of patches by reducing the amount of human effort required to analyze a patch for its trustworthiness. The invariant analysis effort was led by the Arizona State University team.

3.7. Models and Formal Proofs

We also consider formal methods for ensuring trustworthiness of a system. Through the use of modeling and automated proof techniques, we can make guarantees about the state of the system. We automatically generated proofs of the prototype system, allowing us to conclude that the trusted controller could always retain physical control of the rover if an attack was detected against the locomotion controller. Formal architectural proofs were produced by the Kestrel team.

4. Results and Discussion

Our prototype system consists of a dual-controller autonomous rover platform. The locomotion controller executes a given mission using the ArduPilot vehicle control software while the trusted controller monitors the vehicle's behavior for anomalies. If an anomaly occurs, the trusted controller takes control of the vehicle, navigates to a safe location, and automatically synthesizes a repaired version of the control software. When a repair is synthesized, it is deployed on the locomotion controller, at which point it can resume the mission without being affected by the original attack or defect.

We successfully integrated (1) software hardening via Zipr (Section 4.2), (2) runtime monitoring via CMT (Section 4.3), and (3) automated repair via Darjeeling (Section 4.4) on a physical rover prototype. Additionally, we investigated techniques for (4) evaluating the quality of automated repair using invariant detection (Section 4.5) and (5) modeled and proved properties of our prototype's architecture (Section 4.6). Our prototype was evaluated by an external Red Team furnished by the Government.

4.1. Read Team Evaluation

We first present summary results based on our Red Team evaluation. The Red Team provided fourteen seeded defects for our software simulation of ArduPilot, ten of which applied to the physical rover prototype. Tables 1 and 2 describe each of these defect scenarios as well as a summary of whether each component of our integration effort successfully detected, mitigated, or fought through each scenario by the end of the project, in software simulation and physical rover, respectively. Additional details about each component are discussed in Sections 4.2 (Zipr), 4.3 (CMT), and 4.4 (Repair).

Table 1. Summary of the first Red Team evaluation (software simulation only) of each component of our integration effort. “All” indicates that the tool behaved correctly (detected or defeated the attack) in all trials; “Some” indicates that the tool behaved correctly in some random trials. This evaluation took place at the halfway mark of the effort; the insights gained from it were used to improve the techniques for the final evaluation.

ID	**Description**	**Zipr**	**CMT**	**Repair**
1	Use after free	All	All	All
3	Format string → information leak		All	
4	Format string → crash	All	All	
5.1	Stack-based buffer overflow (ground control)	All	All	All
5.2	Heap-based buffer overflow (ground control)	All	All	All
7	Stack-based buffer overflow (MAVLink)	All	All	All
8	Heap-based buffer overflow (MAVLink)		All	All
9	x86 Code Injection		All	All
10	Infinite Loop		All	All
12	Segmentation Fault	All	All	All
13	Mathematical Logic Bug	Some	All	Some
14	Denial of Service		All	
15	Integer Overflow	All	All	All
16	Floating Point Exception	All	All	All

Table 2. Summary of the final Red Team evaluation on physical rover prototype. This represents the final, observed performance of the Trusted and Resilient Mission Operation (TRMO) system. (Note that not all software simulation attacks from Table 1 applied to the hardware prototype. Every applicable attack is shown.)

ID	**Description**	**Zipr**	**CMT**	**Repair**
1	Use after free		Detect	Repair
7	Stack-based buffer overflow (MAVLink)	Detect	Detect	Repair
10	Infinite Loop		Detect	Repair
12	Segmentation Fault	Prevent	Detect	Repair
15	Integer Overflow		Detect	Repair
H1	Integer cast error leading to exception		Detect	Repair
H2	Stack-based buffer overflow		Detect	Repair
k	Arc injection corrupting global structure	Detect	Detect	Repair
g	Stack-based buffer overflow	Detect	Detect	Repair
de	Faulty input sanitization corrupts pointers		Detect	Repair
do	Double free of heap pointer		Detect	Repair

4.2. Software Hardening via Binary Rewriting (Zipr)

Zipr works by accepting a program binary as input (e.g., an executable file) and composing a variety of different transformations to that input, culminating in a hardened output binary that retains the original program's behavior while reducing opportunities for attackers to compromise it at runtime. Additionally, Zipr is capable of providing robust software hardening facilities via transformations that do not incur significant amounts of runtime overhead.

Figure 1 provides a high-level overview of the Zipr architecture and its transformation pipeline. Starting with the original binary, Zipr first performs an initial reverse engineering pass (IR Builder) and populates the Intermediate Represent Database (IRDB) with a state representation of the binary. This representation includes information such as instructions (both in binary and disassembled form), potential indirect branch targets, control-flow information and function boundaries. Transformations are encoded via Zipr plugins (T_x in Figure 1). Zipr provides a simple Application Programming Interface (API) for plugin writers to export the IRDB state into high-level constructs, e.g., instructions, functions, control-flow graphs, manipulate these constructs to implement the desired transformation, and then export the transformed state back into the IRDB. This architecture provides an easy way to compose transformations simply by chaining plugins one after another. To produce the final transformed binary, Zipr extracts the final state from the IRDB and lays out the code and data accordingly.

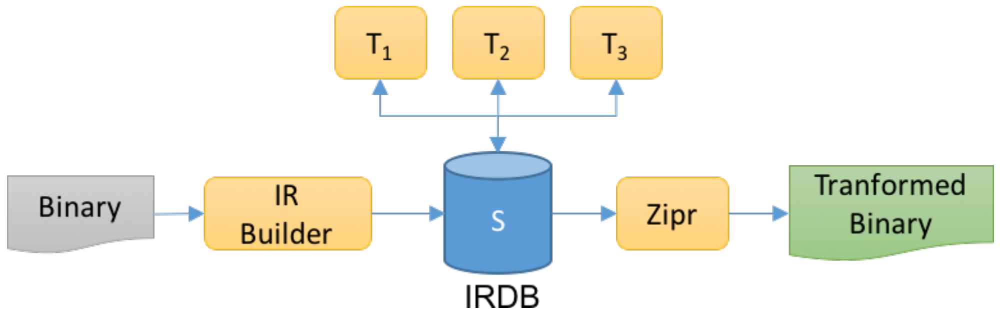

Figure 1. Zipr transformation pipeline for analyzing and transforming

By using a standard SQL database to represent program state, Zipr provides a flexible and modular architecture. Plugins may be expressed in any programming language provided they import/export data using the Zipr schema. The IR builder may be easily replaced with other 3rd-party tools (we have used both an IDA Pro and an internal IR builder).

To our knowledge, Zipr is the only toolchain that enables such a powerful combination of hardening and diversity techniques using only binaries as input.

4.2.1 Red Team Evaluation and Zipr

For Scenarios 3, 4, 5.1, 5.2, and 7 the Zipr defense resulted in exploits being rendered harmless without causing the rover software to terminate. We attribute this improvement to the Binary Auto Repair Template (BinART) transformation that was deployed during this project. BinART leverages compiler-introduced functions that replace potentially unsafe C library functions with safe counterparts (e.g., strcpy_chk instead of strcpy). BinArt replaces the default behavior when these safe functions detect violations with code to tolerate potential errors. For example, if the bounds for a memory copying operation are known, BinArt allows the copy, but limits the length of the target buffer. In cases when the compiler configuration does not generate safe library functions, we can perform static analyses to propagate (when possible) bounds information and rewrite the binary to replace unsafe functions with their safe counterparts.

Some of the Red Team-provided scenarios are beyond the scope of Zipr's static defenses. Of all the in-scope scenarios, Zipr failed to detect the attack in scenario 9 (x86 injection). Scenario 9 overwrites a function pointer. The exploit is deemed successful when a specific function seeded by the Red Team is called. However, the scenario implementation prints out the address of the target function, which limited our ability to relocate the code. Attacks in the wild typically analyze the original binary to retrieve the address of the function (instead of printing it). In such a case, our diversity technique would have moved it, and our Control Flow Integrity (CFI) technique would likely have prevented the control flow transfer.

Finally, Zipr's transformed versions of the scenario binaries provided by the Red Team did not incur significant or measurable runtime overhead—the prototype rover remained as responsive and functional post-transformation as pre-transformation.

4.3. Runtime Monitoring (CMT)

We applied CMT-style continuous monitoring capabilities to the x86 ArduPilot-based autonomous vehicle control software to detect when the operation could be trusted and when its behaviors deviate from expected behavioral ranges. Unlike our earlier work, in this evaluation all of the trust assessment logic was resident onboard the second trusted controller rather than being split between a ground control station and the vehicle. In this setup, runtime monitoring components were then used to cross check externally visible behaviors and states (e.g., the vehicle's location, locomotion, altitude) and to monitor the control software's execution via instrumentation (e.g., external process health monitoring and embedded logical assertions over select command and control access patterns). Critically, this allows CMT to monitor a CPS without incurring networking or storage overhead.

Upon detecting anomalous behaviors, runtime trust violations were remotely reported to the trusted controller for all scenarios that the Red Team developed (Tables 1 and 2). This operation served as a *key trigger* to initiate remedial repair behaviors (i.e., harden, attack, detect, repair, repeat, but remediate the attack step). To support useful trust reporting, we also extended our instrumentation and reporting to provide actionable diagnostic information leading up to the trust violation (i.e., call stacks, exit codes).

Finally, throughout the project, while integrating these technologies, we supported validation, experimentation and demonstration. During both mid-term and the final Red Team evaluation, we validated the effectiveness of our integrated system, both in simulation and on a live rover. We also developed Graphical User Interface (GUIs) for visualizing the progress of the mission, planning attacks, and providing visual clues about assessment of trust violation triggers.

4.3.1 Red Team Evaluation and CMT

Under this project, we ported and integrated CMT's modeling and runtime trust assessment software to our prototype Intel UP rover platform. For this new platform, we showed that it is possible to record and model telemetry, and to cross-check the ArduPilot software's execution on Intel architecture. We also demonstrated that it is possible to run the cross-checking capabilities of CMT live onboard an Intel UP-based autonomous vehicle controller, which was previously a feature limited to the ground control station. This indicates that CMT is able to run successfully in resource-constrained environments, incurring a small amount of overhead.

At the execution-level on the Intel Rover, we developed (1) an ArduPilot process monitor that rapidly detects process exits of the ArduPilot software on the locomotion controller, and (2) logical assertion guards to restrict access to the rover's configuration set via legitimate network communication. While the former development was tested in Red Team evaluation and successfully detected a number of Red Team attack scenarios, the latter development was not tested during evaluation. However, we independently verified that it may be used to guard against certain types of attacks (e.g., direct "return to C library" attacks) where the attacker only has limited access to the binary's call graph. In brief, such techniques effectively guard against a wide swath of attacks against CPSs, thus providing increased trust without significantly impacting runtime performance.

At runtime, CMT cross-checks the in-mission behavior of our prototype vehicle against the modeled constraints supplied by the operator. Decreases in trust levels were then reported as an alert to the trusted controller, which took control of the vehicle and engaged our repair mechanism (Section 4.4). Where possible, these alerts also included actionable information, such as call stacks, process exit signals, and GPS coordinates leading up to the anomaly. We showed that it is possible to collect varying degrees of useful localization information to support automated repair, both with and without the Zipr technology (Section 4.2).

In both the final demonstration and Red Team evaluations, the resulting trust assessment capabilities were shown to detect all anomaly scenarios for simulations and live tests, as shown in Tables 1 and 2.

4.4. Automated Software Repair (Darjeeling)

We adapted automated program techniques to the embedded Intel UP board on the trusted controller of our prototype rover. In brief, when a trust violation occurs, the trusted controller takes physical control of the rover and guides it to a safe location, where the trusted controller uses our repair algorithms to construct a repaired version of the ArduPilot software that eliminates the vulnerability that led to the trust violation. In this section, we discuss other aspects of our repair technique, including performance and efficiency.

4.4.1 Red Team Evaluation and Darjeeling

Overall, we successfully repaired 10 out of 14 vulnerabilities seeded by the Red Team in software simulation and repaired all vulnerabilities in our physical rover prototype. While Tables 1 and 2 indicate that Darjeeling successfully constructed resilient repairs to most of the simulation-based Red Team scenarios and all of the hardware-based Red Team scenarios, we also discuss results demonstrating Darjeeling's efficiency owing to our work to decrease search space during repair.

Search space size, or the number of potential candidate patches, is a key determinant of repair efficiency. Since each element of the space takes time to evaluate, reducing the size of the search space reduces the expected time to produce a repair. The following reports the number of candidate transformations removed by each category of optimization we developed during the project, together with the overall reduction in the total number of transformations:

- No optimizations, original full search space: 43,992
- Ignore string-equivalent code snippets: 43,296 (-626)
- Scope checking for repaired variables: 4,772 (-39,150)
- Keyword scope checking: 43,049 (-873)
- Only construct repair insertions from executed code: 21,188 (-22,734)
- Ignore dead code: 18,508 (-25,414)
- All optimizations: 3,000 (-40,922) – an order-of-magnitude improvement in repair time

Table 3 compares the time taken for Darjeeling to its first repair with our optimizations enabled. On average, optimized Darjeeling finds its first acceptable repair after 107 seconds.

Table 3. Summary of the times to find the first repair per Red Team scenario and the number of patches found per scenario within 15 minutes.

ID	Description	Time to Repair (s)	Acceptable Patches
1	Use after free	38	17
5.1	Stack-based buffer overflow (ground control)	98	10
7	Stack-based buffer overflow (MAVLink)	65	18
10	Infinite Loop	79	3
12	Segmentation Fault	40	6
13	Mathematical Logic Bug	201	6
15	Integer Overflow	35	17
16	Floating Point Exception	297	6

Overall, Darjeeling proved an effective means to automatically synthesize repaired CPS software, enabling resilient system operation in the presence of attackers. Additionally, we successfully demonstrated our ability to synthesize a patch for every Red-Team provided scenario on our physical rover platform within 15 minutes.

4.5. Invariant Analysis

Table 3 shows that multiple candidate patches were produced for each attack. While any patch may allow the vehicle to fight through the attack and continue the mission, in the long term some patches may be more desirable than others (e.g., more trustworthy, easier to maintain, etc.). Our invariant analysis helps the operator to assess and trust candidate patches.

We considered as input multiple patches produced by Darjeeling for each Red Team scenario. We execute each patch using the Valgrind run-time instrumentation tool, which generates trace data that we feed to the Daikon invariant inference algorithm to dynamically learn program invariants. That is, we generated a set of program invariants for each patch synthesized against a single Red Team scenario. We then clustered these sets of program invariants together based on similarity. We then rank-ordered these clusters based upon how many lines of code the original patch required (smaller patches involve less code churn and can be easier to analyze and trust). These clusters enable more efficient trust assessment of patches by reducing the number of individual patches to be considered by a human analyst. The operator need only inspect one patch per cluster, since all patches in the same cluster have the same invariants (behavior). All told, this technique reduced the number of human evaluations by a factor of up to five.

4.6. Models and Formal Proofs

We created a formal model of the dual-controller architecture in the ACL2 theorem prover. We modeled the locomotion controller and the trusted controller as state machines, some of whose transitions are synchronized. The state of the system as a whole consists of a state for the locomotion controller and a state for the trusted controller. A transition of the system consists of a transition for each sub-machine, except that transitions that violate the synchronization constraints are not allowed. We then formalized the key correctness property—that exactly one controller is controlling the system (operating the actuators) at any time—and formally proved the property using ACL2.

We also reviewed and analyzed in more detail both the software simulation and physical rover versions of our integration, focusing on architectural properties. We created a variety of semi-formal models (e.g., sequence diagrams) of the systems. We built a state machine model of the system (visualized in Figure 2) and proved the property discussed above as an ACL2 theorem. The proof was highly automatic. By using formal modeling techniques, we increased the trustworthiness of our integration by proving that the trusted controller can retain physical control of the vehicle if an attack takes place on the locomotion controller. Such formal techniques can be applied more broadly to other CPS applications.

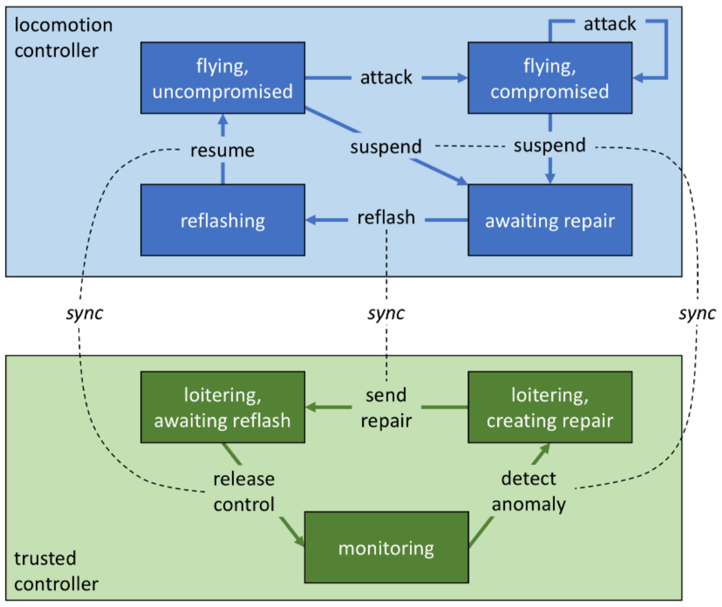

Figure 2. State machine model of our integrated dual-controller architecture. The formal proof that exactly one controller is operating the actuators at a time was made against this model.

5. Conclusions

Cyber physical systems (CPS) form a ubiquitous, networked computing substrate, which is increasingly essential to our nation's civilian and military infrastructure. These systems must be highly resilient to adversaries, perform mission critical functions despite known and unknown vulnerabilities, and protect and repair themselves during or after operational failures and cyber-attacks. We believe that an automated CPS repair approach that can prevent failures of related, mission-critical systems is a necessary component to support the resiliency and survivability of our nation's infrastructure. We integrated and evaluated previously-developed techniques to cooperatively eliminate certain software security vulnerabilities in cyber physical systems, to repair certain general classes of such systems, and to increase the confidence of human operators in the trustworthiness of the repairs and the subsequent system behavior. We worked with a Government-provided Red Team to demonstrate and validate our approach on embedded platforms, including an autonomous rover vehicle.

We first considered transformations to software that reduced opportunities for exploitation while preserving the software's original behavior, thus increasing the trustworthiness of the system overall. Through the application of our Zipr toolchain, we successfully applied a wide range of diversity and hardening transformations to CPS software that do not require the availability of source code or debugging symbols, which is common in commercially-available software. The ability to work directly and efficiently without source code is a hallmark feature of the Zipr toolchain that distinguishes it from other approaches to hardening and diversifying software, allowing us to increase trust in systems where source code is unavailable.

Second, we used runtime monitoring and verification to maintain trusted and resilient operation of cyber physical systems. We applied and developed trust verification concepts and runtime monitoring capabilities as part of an integrated platform that detects signals that indicate a decrease in operator trust. Our Composable and Measurable view of Trust (CMT) tool has been enhanced and integrated into our prototype platform, detecting trust violations that result from software defects and malicious activity. These trust violations can feed into other aspects of our integrated platform to guide automatic repair and software hardening, providing resilient cyber physical systems in spite of anomalous or malicious activity.

Third, we employed automated program repair techniques to synthesize new versions of CPS software that are immune to the software defect leading to the observed anomalous or malicious behavior. Our Darjeeling tool, a modern implementation of GenProg algorithms, synthesizes repairs via software mutation. This can provide significant resilience in the face of security attacks and latent software engineering defects. However, the effect and merit of such repair actions may not be obvious to the human operators who must ultimately make deployment decisions. As a result, supporting trust in systems that make use of automated program repair is an ongoing research question. To improve trust in software, we gathered multiple modalities of evidence (including statistical evidence and formal invariants). This evidence can be presented to human operators; at a high level it answers questions such as "in what ways is this system similar to systems I already trust?" and "how can I characterize the properties that will hold as this system executes?"

Fourth, we used formal descriptions of normal CPS program behavior to cluster these synthesized repairs based on functional changes. This can be leveraged to reduce the amount of human overhead in the problem of patch validation: choosing which of several candidate solutions to apply. In this project, we examined defects in the ArduPilot codebase and several corresponding repairs we synthesized. We clustered these repairs based on the invariants inferred after the repair was applied. This significantly reduced the number of patch evaluations required by a human, thus allowing us to more quickly evaluate the quality and trustworthiness of patches.

Finally, we analyzed our integration effort and created formal and semi-formal models of the prototype to help increase trust. We created a formal model our integrated prototype platform, represented in the language of the ACL2 theorem prover. We then used ACL2 to prove a key property of this model: that we can retain physical control of the prototype in spite of an attack.

In summary, we developed a dual-controller autonomous vehicle architecture integrating multiple software hardening, diversity, and repair techniques together, forming a coherent, viable prototype autonomous vehicle. We worked with a Government-provided Red Team to evaluate our integration effort both in software simulation and on a physical rover platform. We successfully detected, mitigated, and repaired 10 out of 14 attacks provided by the Red Team in an early software simulation. *We successfully detected, mitigated and repaired and all attacks during the final evaluation on the physical rover platform.* We also produced a live demonstration showcasing our combination of techniques providing trust and resilience in autonomous vehicle missions.

6. Appendix A: Publications

Joseph Renzullo, Stephanie Forrest, Westley Weimer and Melanie Moses: **Neutrality and Epistasis in Program Space**. In *Genetic Improvement*, 2018 (best presentation award)

Christopher Steven Timperley, Afsoon Afzal, Deborah S. Katz, Jam Marcos Hernandez, Claire Le Goues: **Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early?** In *11th IEEE International Conference on Software Testing, Verification and Validation.* ICST 2018: 331-342

Mauricio Soto, Claire Le Goues: **Using a probabilistic model to predict bug fixes.** In *25th International Conference on Software Analysis, Evolution and Reengineering.* SANER 2018: 221-231

Mauricio Soto, Claire Le Goues: **Common statement kind changes to inform automatic program repair.** In *Proceedings of the 15th International Conference on Mining Software Repositories.* MSR 2018: 102-105

Christopher Steven Timperley, Susan Stepney and Claire Le Goues. **Poster: BugZoo – A Platform for Studying Software Bugs.** In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering*. ICSE 2018

William H. Hawkins, Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Jack W. Davidson: **Securing Binary Code.** In *IEEE Security & Privacy* 15(6): 77-81 (2017)

7. List of Symbols, Abbreviations, and Acronyms

ACL2: A Computational Logic for Lisp, an automated theorem prover

API: Application Programming Interface

BinART: Binary Auto Repair Template

CFI: Control-Flow Integrity

CMT: Composable and Measurable Views of Trust

CPS: Cyber-Physical System

Daikon: A tool that detects likely invariants of a program via dynamic analysis

Darjeeling: Generic approach to language-agnostic program repair

GUIs: Graphical User Interface

IRDB: Intermediate Represent Database

MAVLink: Micro Air Vehicle Link, a protocol for communicating with autonomous vehicles

TRMO: Trusted and Resilient Mission Operation

Valgrind: An instrumentation tool used to gather information about program execution

Zipr: A binary-rewriting tool capable of transforming arbitrary software programs