AFRL-RI-RS-TR-2020-095

TRUSTED AND RESILIENT MISSION OPERATIONS

COMPUTER SCIENCE & ENGINEERING
REGENTS OF THE UNIVERSITY OF MICHIGAN

JUNE 2020

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

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

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

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. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2020-095 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILMAR SIFRE
Work Unit Manager

/ S /
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUNE 2020	FINAL TECHNICAL REPORT	SEP 2017 – FEB 2020

4. TITLE AND SUBTITLE

TRUSTED AND RESILIENT MISSION OPERATIONS

5a. CONTRACT NUMBER
FA8750-19-2-0006

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)
Westley Weimer
Jack Davidson
Stephanie Forrest
Clair Le Goues
Partha Pal
Eric Smith

5d. PROJECT NUMBER
T2EE

5e. TASK NUMBER
MI

5f. WORK UNIT NUMBER
CH

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science & Engineering
Regents of the University of Michigan
Office of Research and Sponsored Programs
3003 S. State Street, Ann Arbor, MI 48109-1274

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-2020-095

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

The Trusted and Resilient Mission Operation (TRMO) effort developed, integrated, and evaluated software techniques for the trusted, resilient operation of a diverse set of vehicular cyber physical systems. Activities focused on resilient approaches that apply to multiple off-the-shelf architectures, allowing an uncrewed autonomous vehicle to demonstrate robust defensive measures against would-be software-based attackers.

15. SUBJECT TERMS
Automated program repair, binary hardening, software diversity, anomaly detection, formal proof, software invariant, red team.

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	71	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Contents

List of Figures

List of Tables

1 Summary

The *Trusted and Resilient Mission Operation* (TRMO) effort developed, integrated, and evaluated software techniques for the trusted, resilient operation of a diverse set of vehicular cyber physical systems (CPS). Activities focused on resilient approaches that apply to multiple off-the-shelf architectures, allowing an uncrewed autonomous vehicle (UAV) to demonstrate robust defensive measures against would-be software-based attackers.

TRMO uses a combination of best-in-breed static and dynamic methods to create such a trusted, resilient system. Before deployment, TRMO employs hardening, diversifying and rewriting techniques to create, select, and deploy defensive variants of existing control software. In addition, TRMO analyzes and models the correct behavior of the system and its invariants, establishing a formal notion of trusted execution. TRMO separates trust monitoring and resiliency actions from payload software; this can be done by leveraging existing secure operating systems or via a dual controller architecture. When a trust policy violation is detected, a software repair is constructed with respect to learned invariants and software simulations of indicative missions. Automated software repairs increase system resiliency, allowing missions to continue in the face of software defects and adversarial exploitation. During deployment, including after any repairs, the system is continuously monitored.

The activities carried out under this effort include the following:

1. Demonstrating resilience and technological maturity on a commercially-available Intel x86-based ground rover as well as a commercially-available ARM-based quadcopter.
2. Validating TRMO's capability to support during-mission repairs to control software, with resiliency actions completed in a five- to ten-minute timeframe.
3. Assessing the effectiveness of TRMO via multiple live demonstrations against indicative cyberattacks furnished by a government-provided Red Team.

The research and evaluation activities described in this report span multiple years of project effort during the Trusted and Resilient Systems program. TRMO brings together successful academic and industrial researchers and ideas. These include repair techniques explored as part of a MURI project, software diversity approaches showcased under the Helix[1] and Cyber Fault-tolerant Attack Recovery[2] (CFAR) programs, and defensive technologies demonstrated at the Cyber Grand Challenge (CGC), among others. Team expertise includes formal methods (Kestrel Institute), adaptive systems (Arizona State), binary defense (Virginia), program analysis and repair (Carnegie Mellon and Michigan), and trust and integration (BBN Raytheon).

Ultimately, this report describes the development and measurement of a defensive system in which vehicular control software is diversified, analyzed, and modeled before deployment in a multi-controller or off-the-shelf system. Hardening and learned safety envelopes admit the detection of certain classes of cyberattacks and anomalies at runtime. A resiliency subsystem constructs repairs, which are verified against key invariants and deployed mid-mission. The proposed techniques apply to both Intel-based and ARM-based rovers and quadcopters. Evaluation metrics and success criteria are evaluated on indicative attack scenarios furnished by a Red Team. In the final Red Team evaluation of these defenses applied to 21 attacks against an ARM quadcopter, TRMO detected 90% of attacks. In addition, TRMO constructed a repair for 62% of attacks, allowing the vehicle to fight through and complete the mission successfully.

[1]Results produced under government contract FA9550-07-1-0532 and publicly available.

[2]Results produced under government contract FA8750-15-C-0118 and publicly available.

Approved for Public Release; Distribution Unlimited.

2 Introduction

The growing ubiquity of uncrewed autonomous systems has resulted in a wealth of interest from the defense and academic communities. The Secretary of Defense has provided a roadmap for the development of these systems, which goes beyond vehicles to include a specific focus on ground-based sensors, processing, and communication [61].

There is increasing demand for systems that are both trusted (e.g., [22]) and resilient (e.g., [64]) in the face of unanticipated challenges. In this context, a *resilient* system recovers from — or avoids — errors, attacks, trust policy violations or environmental challenges to complete its original mission or a variant thereof [31]. A *trusted* system is one in which human operators have confidence in the correct operation of the system, even if the resiliency actions change its software or configuration during deployment.

One approach to resiliency anticipates possible problems a system could encounter and devises pre-programmed responses. The DARPA High Assurance Cyber Military Systems (HACMS) [22] program, which synthesizes control software for various uncrewed platforms, exemplifies this approach. Formal methods can provide trust through correctness, safety, and security guarantees, but may not provide adaptive resiliency.

A second approach to providing resiliency focuses on adaptability through automated program repair, binary rewriting, or runtime monitoring. Such black-box-guided adaptations do not necessarily guarantee correctness, but often allow systems to automatically overcome errors or attacks, have demonstrated resiliency to unknown attacks [5, 23, 46] and defects [43], and may sometimes be partially validated post hoc.

The current state of the art, even for cyber-physical systems (CPS), is manual debugging, verification and validation (e.g,. [76]). A detected defect must be localized (e.g., [27]), and developers must understand both the implementation and specification to identify changes to bring the current implementation closer in line with the desired behavior. This process is time-consuming and expensive.

This report describes the *Trusted Resilient Mission Operation* (TRMO) system architecture, which focuses on efficient, trusted, resilient techniques that allow autonomous vehicles to "fight through" an attack and continue a mission, possibly in a degraded mode of operation. This work integrates a set of static and dynamic protective technologies for Intel- and ARM-based rover and quadcopter platforms. This report presents evaluations of the maturity and technical readiness our of techniques by applying them to extensive simulations and live demonstrations against Red Team-provided cyberattack scenarios.

Technical Summary. This effort developed, integrated and evaluated techniques to provide trusted and resilient operation for autonomous vehicle missions via a combination of static defenses, models and proofs as well as dynamic monitoring and repair. In the final Red Team evaluation using an ARM quadcopter, TRMO detected 90% of attacks and constructed a repair for 62%, allowing the vehicle to fight through and complete those mission successfully.

2.1 Background

The relevance of uncrewed autonomous vehicle systems and the desire for simultaneous trust and resilience in such systems are central to this effort. We validate our approaches using ground-based rover and aerial quadcopter exemplar systems.

3 Methods, Assumptions, Procedures

The Trusted and Resilient Mission Operation system combines modeling, formal proof and hardening with dynamic detection and repair actions to provide trusted, resilient autonomous vehicle operation. We focus on readily-available commercial off-the-shelf (COTS) autonomous vehicles, including both ground-based rovers and aerial quadcopters, assuming access to locomotion control software and the capability to add additional COTS hardware or leverage a secure operating system.

3.1 Technical Approach

Figure 1 presents a high-level overview of the TRMO system architecture.

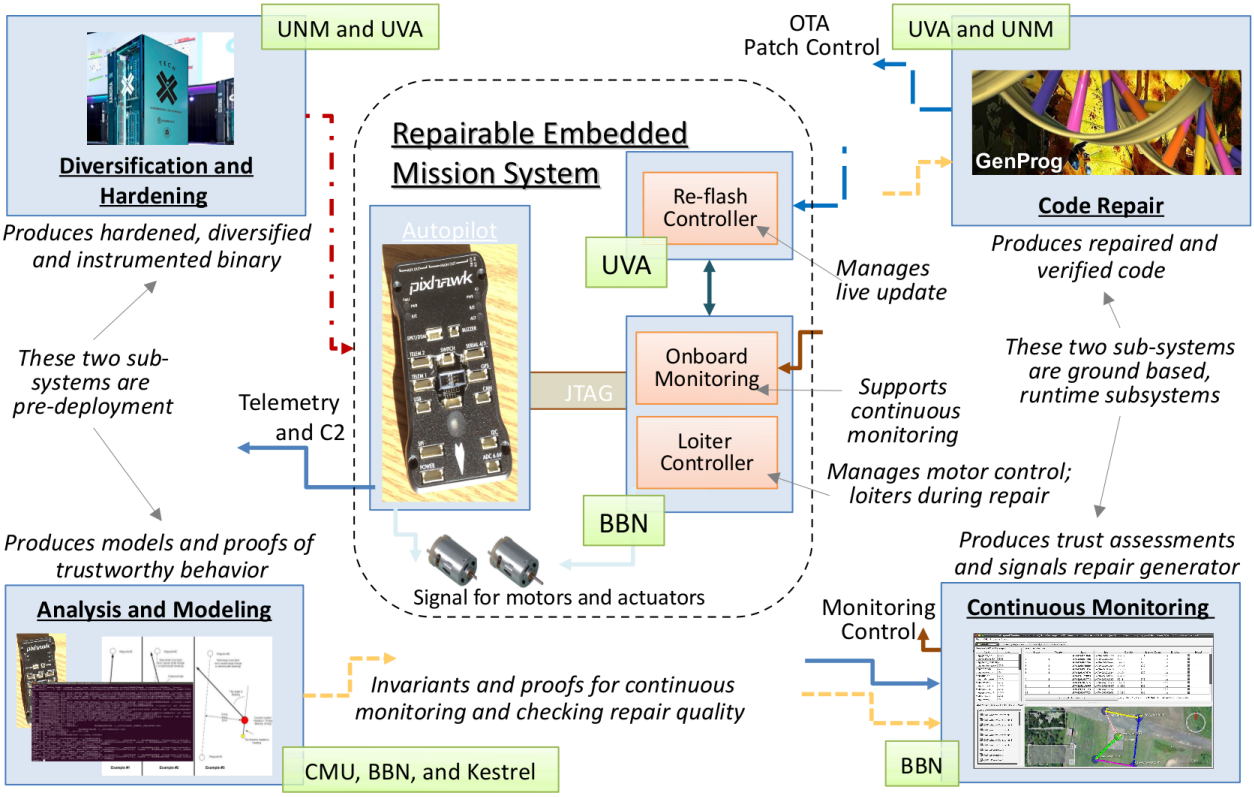

Figure 1: Trusted and Resilient Mission Operation system architecture. Activities shown on the left (Diversification and Hardening, Analysis and Modeling) are carried out before the mission. During the mission, activities on the right (Continuous Monitoring, Code Repair) are performed. The result is a deployed system that provides resiliency against certain classes of cyberattacks while admitting operator trust. Team expertise and enabling technologies are highlighted.

In summary, the four primary components include:

- Diversification and Hardening. Given indicative missions, we analyze and **harden** the locomotion control software. At the static binary level, we harden software to defeat certain classes of attacks (Section 3.3). Dynamically, we learn trusted envelopes for critical telemetry and runtime values (Section 3.4). At the static source code level, we learn key program invariants that encode partial aspects of correctness (Section 3.6).

- Continuous Monitoring. During **deployment**, a second trusted controller or secure operating system is included with supervisory access to sensors and actuators. It detects mission anomalies and policy violations (Section 3.4), and keeps the vehicle in a safe state while applying platform-agnostic repair actions to the main controller.

- Code Repair. When a trust policy violation is detected, we use program-level techniques to synthesize an adaptive **repair** action (Section 3.5). This includes dynamic evidence that the post-repair mission remains within acceptable parameters (Section 3.4).

- Analysis and Modeling. In addition to dynamic resilience and measurement-based trust, we employ **formal proof** techniques to provide **trust** in aspects of our system (Section 3.6).

3.2 System Assumptions

TRMO makes a number of assumptions about available resources, the scope of the system to be defended, and the threat model describing incoming attacks.

First, TRMO assumes that the *source code* for the locomotion control software to be defended is available. Many TRMO components can lift this assumption and operate on binaries, including Diversification and Hardening, Code Repair, and Continuous Monitoring. However, certain other activities, such as learning formal invariants that describe correctness, require access to the source code. If the source code is not not available the resiliency provided by TRMO remains unchanged but the evidence presented in support of operator trusted is weakened.

Second, TRMO assumes a *threat model* limited to a remote attacker exploiting a software vulnerability. Signal jamming, GPS spoofing, and physical attacks are out of scope. We explicitly consider and defend against a sensor-based attack during the evaluation, but the root cause of the vulnerability was in the software processing the sensor data, rather than the physical hardware. TRMO does not assume any cryptography, and must defend the system even when the attacker has all relevant cryptographic keys or to has otherwise bypassed encryption.

Third, TRMO assumes a *secured subcomponent* from which to conduct analysis, modeling and repair. This can either take the form of additional hardware with a more limited connection to the outside world (a "dual-controller" solution we investigate and evaluation) or it can take the form of an existing secured operating system, such as seL4 [66], from which monitoring and repair actions can be carried out. This work considers the integrity of the monitoring and repair components to be an orthogonal problem, and direct attacks against them are out of scope.

3.3 Diversification and Hardening Component

The TRMO system architecture provides hardening and diversification capabilities via the *Helix* family of technologies. The purpose of hardening and diversification is twofold. First, hardening can entirely defeat certain classes of attacks, such as those based on certain low-level security

exploits. Second, diversification can present a shifting attack surface: necessary parameters or details learned by an attacker can be invalidated, increasing the work required to mount the attack to impractical levels.

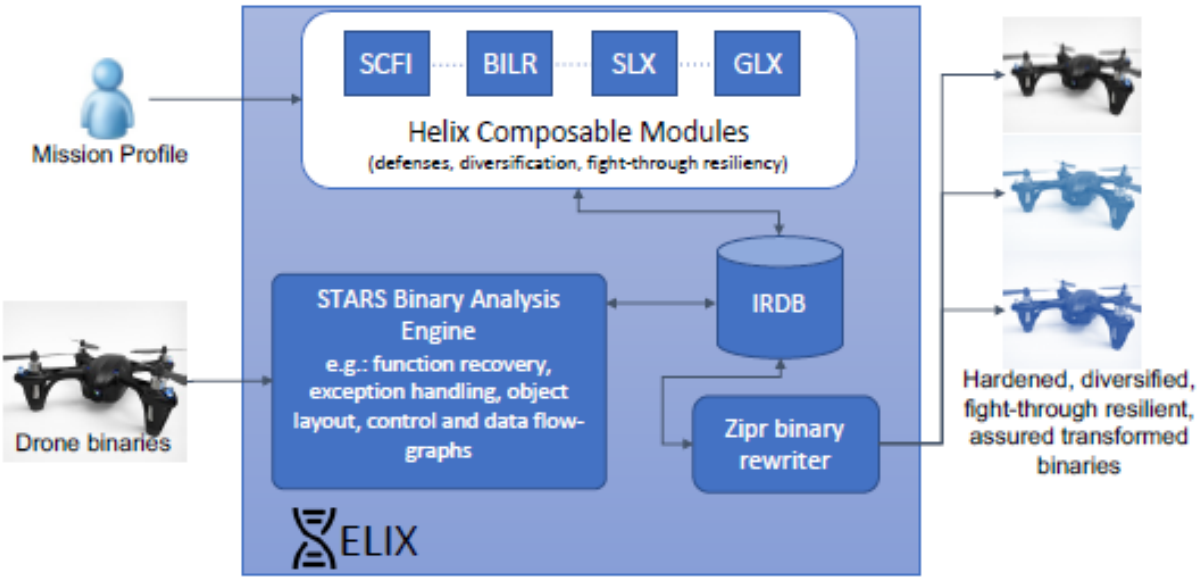

Figure 2: Helix architecture for transforming UAV binaries. Given a mission profile and input binary, Helix generates a diversified set of functionally-equivalent and hardened binaries.

The Helix toolchain (see Figure 2) takes as input a binary and outputs a diversified set of functionally-equivalent and hardened binaries [18, 20, 32, 60]. A key central element of the Helix architecture is its Intermediate Representation Database (IRDB). The IRDB stores a representation of a binary and allows modification of this representation via a standard structured query language (SQL) database interface. It also supports a *clone* operation for creating new duplicate versions of this representation; this operation is critical for efficient diversity.

The first stage in the Helix pipeline performs a reverse-engineering pass using STARS Binary Analysis Engine, and inserts the initial representation of the input binary into the IRDB. Helix modules can then effect their desired transformations by modifying the state representation of binaries stored in the IRDB using a high-level application programming interface (API) that provides abstractions for instructions, functions, data and control-flow information. The last stage in the Helix pipeline, *Zipr*, emits an executable binary.

Helix supports the following composable transformations:

- Block-level Instruction Location Randomization (BILR). BILR randomizes the location of 99%+ of the instructions in a binary program. Any attack that relies on knowing the exact location of a code address will most likely be thwarted ($\leq$ 5-10% overhead, x86, ARM).

- Selective Control-Flow Integrity (SCFI). Helix infers the control-flow specification by reverse-engineering the binary and rewrites the binary to enforce the inferred specification at run-time. SCFI thwarts arc-injection attacks, i.e., return-to-lib, return-oriented programming, and blind return-oriented programming attacks ($\leq$ 12% overhead, x86).

- Heap Layout Transformation (HLX). HLX thwarts heap-based attacks, including double-free attacks and attacks that target the control block of the standard heap library or otherwise target the predictable layout of heap-allocated memory ($\leq$ 5% overhead, x86, ARM).
- Stack Layout Transformation (SLX). SLX uses stack canaries to detect stack-smashing attacks and inserts random padding into the stack frame, thereby adding entropy to the stack layout ($\leq$ 10% overhead, x86).
- Global Layout Transformation (GLX). GLX adds randomness to the layout of global variables ($\leq$ 1% overhead, x86).
- Binary Auto-Repair Templates (BinART). BinART provides forward-error recovery capabilities for well-known vulnerability patterns ($\leq$ 1% overhead, x86).

The strength of binary diversification and hardening techniques depends heavily on the accuracy and precision of analysis results. Helixs philosophy can be informally described as a Binary Hippocratic Oath, "thou shall do no harm". As an over-aggressive transformation may result in broken functionality, Helix errs on the side of caution. Thus, in places where analysis must be conservative, Helix will transform with caution (e.g., BILR leaves 1% of instructions in place). In places where high-confidence can be obtained, Helix will be more aggressive (e.g., BILR relocates 99% of instructions).

Table 1: Helix attack class coverage. Composing transformations provides defense-in-depth capabilities.

	BILR	SLX	HLX	SCFI	GLX	BinART	Composed
Code Injection Attacks	●	⊙	⊙	●	⊙	⊙	●
Arc Injection Attacks (ROP, Blind ROP)	●	⊙	⊙	●	⊙	⊙	●
Buffer Overflow Attacks	⊙	⊙	⊙	⊙	⊙	⊙	●
Data attacks	⊙	⊙	⊙	⊙	⊙	⊙	⊙
Double free, use-after-free	○	○	●	○	○	○	●
Legend:	○ **No coverage**	⊙ **Partial Coverage**	● **Strong Coverage**				

To thwart a wide-range of attacks, Helix transformations may be composed arbitrarily, providing defense-in-depth. Successful attacks must often rely on several conjunctive assumptions, e.g., knowledge of a vulnerability point and knowledge of a target address. Invalidating any one[1] of these assumptions is sufficient to thwart the attack. Anticipated attack classes covered by Helix are shown in Table 1.

3.3.1 Helix Deployment Scenarios

One goal of this effort is to provide the government with a robust and easy-to-use platform. The Helix component has matured to the point where composing defenses is a one-line command. To illustrate Helix's ease-of-use, the following command takes as input a binary, `/usr/bin/bc`, and outputs a transformed binary, `bc.helix`, that is augmented with a combination of both hardening and diversity transformations:

```
$HELIX /usr/bin/bc bc.helix -s slx=on -s scfi=on --backend zipr
```

Here the `-s slx=on` option turns on the SLX transformation, the `-s scfi=on` option turns on the SCFI transformation, and the `--backend zipr` specifies the use of static rewriting for laying out a diversified binary. The output binary `bc.helix` can be used as a drop-in replacement for `/usr/bin/bc` as they are functionally equivalent on benign inputs.

In evaluations of TRMO, including both simulations and live demonstration, a Helix-generated flight control program (i.e., `ArduPilot`) binary replaced the original.

System operators may wish to support different combinations of Helix transformations for different binaries. In the tradeoff between security and performance, operators may be willing to emphasize security above all other consideration for highly-critical binaries such as network-facing servers (e.g. ,`sshd, bind, httpd`). A simple scheme for implementing such a policy would be to create different directories to denote different configurations and then modify the default search path to prioritize binaries in a high-security directory over those in a medium-security directory.

The Helix hardening architecture provides additional time and flexibility in the face of certain classes of security of security attacks. Operators may choose to patch on their desired timeline and at their full discretion. Once patched, operators may then re-apply Helix to re-protect and re-diversify patched binaries.

While the primary evaluations of TRMO focus on embedded systems, there is also significant interest in protecting cloud-, container- or virtual-machine backed software assets. In such a setting, the process outlined above could be fully automated (Figure 3). Moreover, by periodically refreshing the production of Helix binaries (with different random seeds), the government gains the capability of deploying a practical and effective *moving target defense*, in which the attack surface of deployed systems is both reduced (Helix hardening transformations) and shifted in time (Helix diversifying transformations).

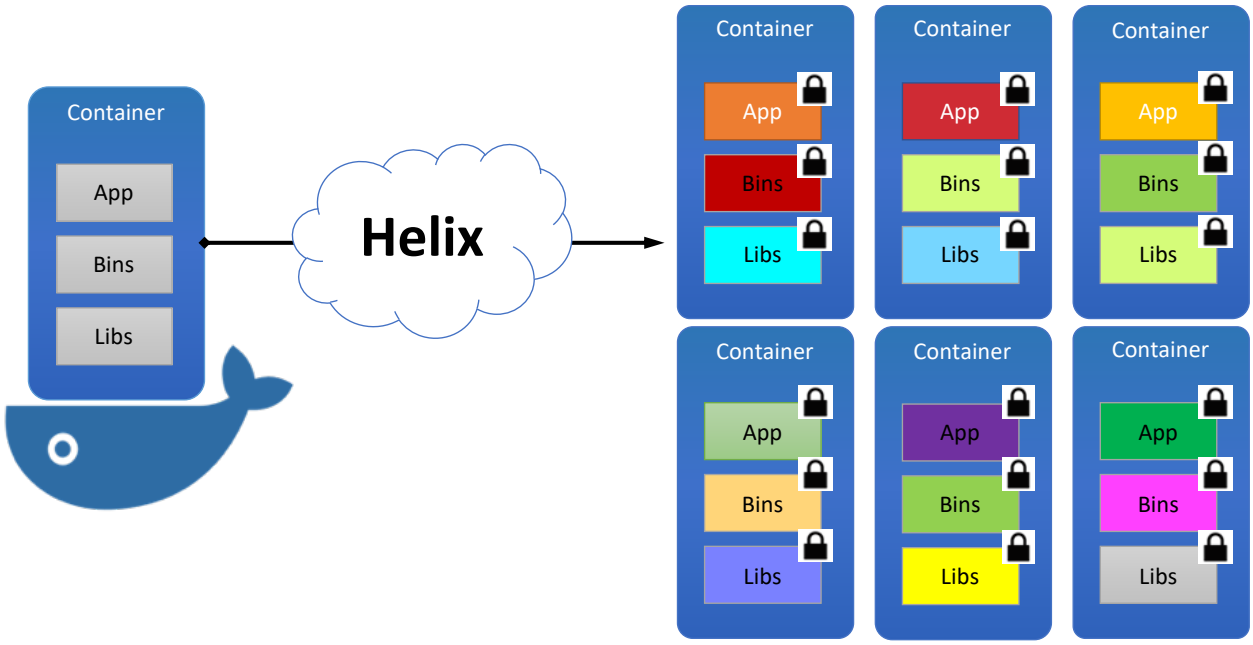

Figure 3: Possible cloud-based deployment scenario: Helix generates diversified and hardened containers.

3.3.2 New Capabilities Enabled by the Helix Platform

Research and evaluation efforts associated with this project resulted in multiple scientific advances associated with diversification and hardening. We briefly report on five: (1) static rewriting for exception handling in languages like C++; (2) "breadcrumbs" that provide information to repair algorithms; (3) binary-level repairs of certain classes of defects; (4) binary enforcement of control-flow integrity; and (5) efficient binary fuzz testing.

Zipr++ [35]. Highly robust, late-stage manipulation of arbitrary binaries to apply systemic changes is difficult due to complex implementation techniques and the corresponding software structures. Indeed, many binary rewriters have limitations that constrain their use. To the best of our knowledge, no static binary rewriters handle applications that use exception handlers or stack unwinding—a feature integral to programming languages such as C++, Ada, Common Lisp, ML, to name a few. Without such support, transformations such as SLX that modify the stack layout, are not reliable and may result in broken programs.

Zipr++, an extension to the base Zipr static rewriter used in Helix, enables the manipulation of applications with exception handlers and tables, which are required for unwinding the stack. Unwinding the stack is necessary for handling exceptions, object-oriented programs (invoking necessary destructors), multi-threading (when a thread exits, destructors on the stack must be invoked), and to support debuggers such as gdb, dbx, and lldb. For example, C++ is the source language for the controller (`ArduPilot`) used in the evaluations of the TRMO system.

Academic papers on binary rewriting often leave support for exception handling information as "future work" and understate the difficulty of actually transforming binaries with exception

handling. To support the community, we have released `libehp`,[3] an open-source library for parsing exception handling tables, which we hope will aid the community in improving static rewriting technology.[4]

Figure 4: Helix lightweight monitoring (breadcrumbs) to inform the TRMO repair process.

Lightweight Breadcrumbs. The notion of lightweight "breadcrumbs" is to augment binaries with the ability to record information as a binary executes. In the event of an attack, such information can be extracted to aid accurate fault identification and localization activities (Figure 4). Such breadcrumbs should be both *generic*, to allow transformations to record arbitrary information, and *lightweight*, so as to not overtly impact normal operation. In the event of an attack, the information contained in these lightweight breadcrumbs can be inspected to guide the repair process (Section 3.5).

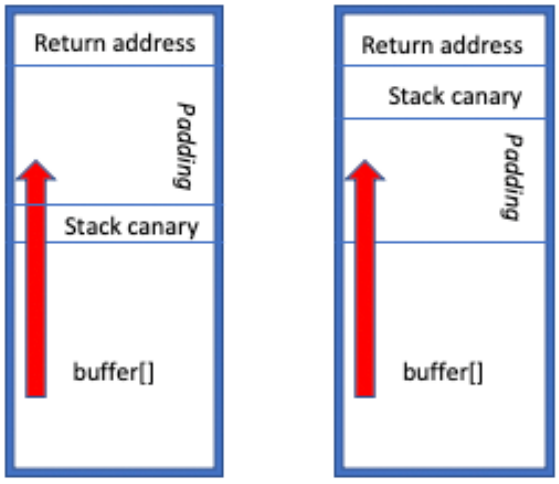

Figure 5: Stack-Layout Transformation (SLX): Relocating stack canary using breadcrumb information. Buffer overflows into padded region without overwriting the relocated canary.

Consider the Stack Layout Transformation (SLX) which pads stack frames and uses stack canaries to detect buffer overflows. A breadcrumb that records (1) the exact location of the canary

[3] https://git.zephyr-software.com/opensrc/libehp

[4] GrammaTech, Inc. is an example of a third-party that has used `libehp` and contributed code

with respect to the stack frame; and (2) the enclosing function, can guide the repair process towards solutions that include a bounds check near the vulnerable function, or a solution where the location of the canary can be moved higher up in the padded region of the stack frame. As a common vulnerability pattern is for a bounds check to be off by a small number of bytes, the approach of relocating a canary may result in an attack to overflow harmlessly into the padded region, and more importantly, allow for continued execution of the binary to achieve fight-through capabilities (Figure 5).

To implement breadcrumbs, Helix provides tool support for transformation writers to record arbitrary information in named global variables. The baseline cost of recording a breadcrumb is near zero, i.e., it is the same as assigning a value to a global variable. In the event of an attack, extracting breadcrumb information simply consists of searching core dumps for the name of the relevant global variables.

Binary Auto-Repair Templates (BinART). Developers often employ unsafe programming constructs when using libraries. Such mistakes are sufficiently common that compilers will often warn about them (e.g., alarming on misused format string APIs or unbounded string manipulation routines). Modern compilers will often substitute unsafe API calls with their safer counterparts. For example, when performing copy operations on buffers with known sizes, some compilers will automatically use the bounded versions of a library call, such as `strncpy` instead of `strcpy`. Moreover, the substituted versions will abort execution when they detect an out-of-bounds violation.

However, aborting execution and fail-stop semantics may result in unwanted software crashes or reduced availability. To enable fight-through capabilities, Helix uses library interposition techniques to intercept and wrap such calls. Whenever possible, instead of aborting execution, BinART attempts to perform forward error recovery techniques, based on the semantics of each API call. For string and buffer operations, Helix will clip copy operations to the specified buffer sizes to prevent out-of-bounds writes. For C-string operations, Helix can be configured to terminate strings with the requisite null byte terminator.

Note that such forward error recovery is not fully semantics preserving. This is the intention: as with all repair techniques (Section 3.5), we desire the protected system's behavior to be different on malicious inputs (in that it should not exhibit the attacker-desired behavior). When using BinART, we rely on extensive regression testing to validate baseline functionality on benign inputs.

Control-Flow Integrity (CFI). *Control-flow integrity* is a powerful technique to enforce the property that the execution of a program conforms to its intended control-flow graph (CFG) specification. Such enforcement thwarts sophisticated arc-injection attacks, including (blind) return-oriented programming attacks [12].

The challenge for Helix and other binary-based approaches is recovering the CFG precisely using only the binary as input, and then enforcing the inferred CFG without incurring onerous performance overheads. During the DARPA Cyber Grand Challenge (CGC), our team employed a lightweight version of CFI and achieved the best defensive score, while staying within an acceptable performance envelope [32, 60].

For TRMO x86 applications, we deployed an enhanced, optimized version of CFI that worked on 64-bit architectures, and resulted in more effective security-performance tradeoffs. This new

CFI achieves higher performance for the same level of security, or higher levels of security for a slight decrease in performance (Section 4.1.5).

ZAFL [33]. *Fuzz testing* is an automated approach to probing for software vulnerabilities, particularly security flaws, by executing a program on random inputs and monitoring for undesired behavior [28, 29]. The robustness and efficiency of Helix and its binary rewriting technology has enabled fast binary fuzzing capabilities. In a separately-funded project, we have developed a Helix plugin to insert fuzzing instrumentation that is compatible with the state-of-the-art American Fuzzy Lop (AFL) fuzzer [1]. AFL is one of the best-known and most effective fuzzing platforms. Its bug-hunting successes include a multitude of severe vulnerabilities uncovered in well-known and widely deployed applications and libraries, including Firefox, LibreOffice, OpenSSL and OpenSSH [1].

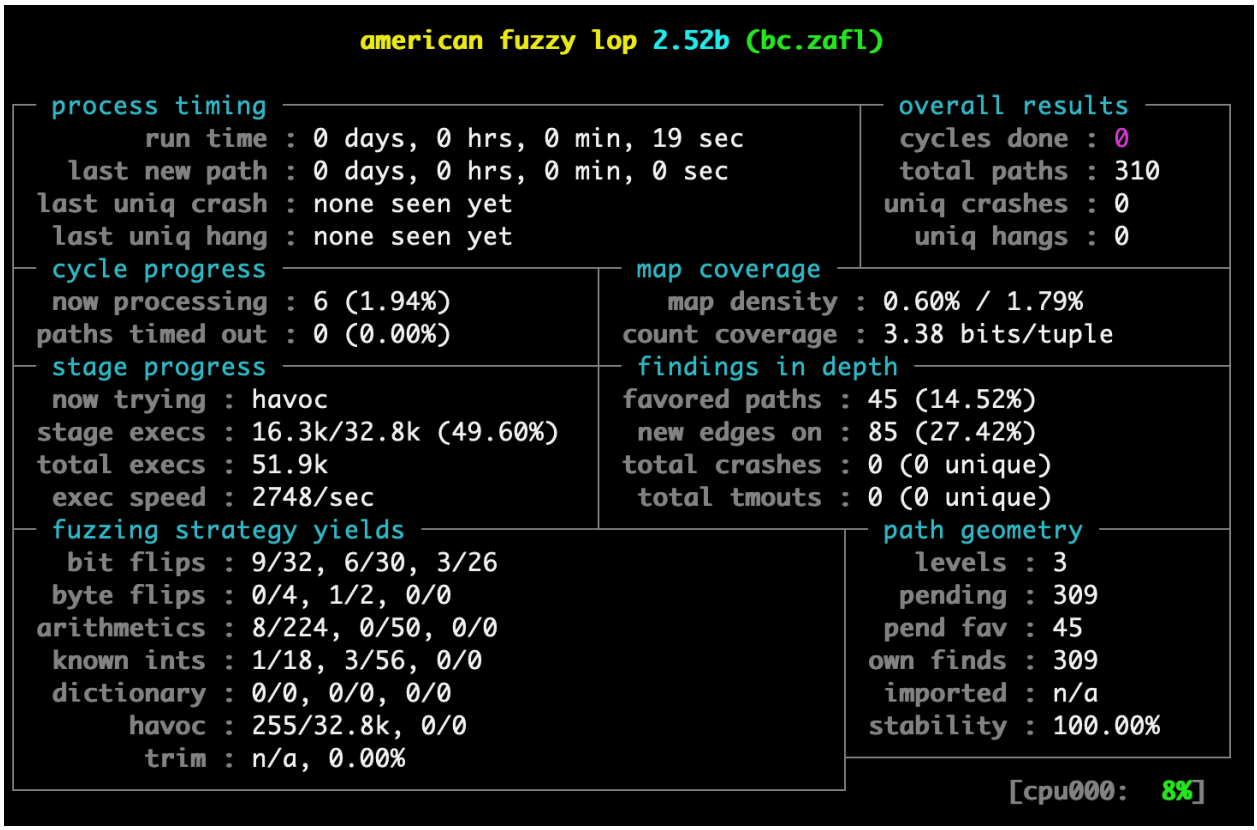

Figure 6: Novel Helix fuzz testing integration. An AFL standard dashboard shows ZAFL compatibility.

The standard AFL workflow consists of modifying the build system to use a special version of the compiler (e.g., clang or gcc), and thus requires source code availability. The modified compiler inserts instrumentation to guide the fuzzing process. Our Zipr-supported binary fuzzing framework, *ZAFL*, inserts the same instrumentation directly into binaries. The resulting binaries are 100% AFL compatible (Figure 6). While AFL provides binary support using QEMU [4], it is highly inefficient. To the best of our knowledge, ZAFL is the *fastest binary fuzzer* available for the AFL ecosystem (Section 4.1.6).

The current TRMO software stack provides proactive diversity and proactive hardening, but reactive repair capabilities. Attackers already use fuzzing techniques to search for exploitable bugs. The further development of ZAFL holds the promise for employing fuzzing towards defensive purposes and enabling *proactive* repair capabilities. ZAFL enables a new paradigm where UAV binaries are actively fuzzed to reveal latent security-critical bugs which can then be repaired by TRMO automatically, prior to mission deployment.

3.3.3 Applying Helix to ARM Systems

Prior to TRMO, the Helix toolchain was supported on x86-32 and x86-64, and was primarily targeted towards laptops, desktops, and servers. As part of the TRMO effort, Helix was expanded to support vehicular, Internet of Things (IOT), and robotics platforms. In particular, we extended the core Helix toolchain to support ARM-32 and ARM-64 platforms:

- ARM instructions are fixed length,
- ARM supports run-time mode switching between ARM and Thumb mode,
- Unlike x86 where the program counter is not directly accessible, ARM provides direct read and write access,
- ARM supports conditional instructions,
- ARM enforces strict code alignment, and
- ARM intermingles data in the code section.

To handle such issues, as well as new issues that will inevitably arise when incorporating a new architecture (e.g., branch delay slots), we refactored the code base for Helix to isolate architectural dependencies, as well as use the Capstone disassembly framework [3] as it supports a plethora of architectures. Insights gained from the TRMO effort allowed the Helix toolchain to be subsequently ported to the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture in only a few days.

3.4 Analysis and Deployment Component

Having described our technical approach to proactive hardening and diversification, in this subsection we describe our technical approach to continuous monitoring and runtime verification. At a high level, the goal is to build a model of trusted system behavior and immediately detect any subsequent deviations from that trusted operating envelope.

We believe trusted mission operations, especially those involving UAVs, must have a strong element of continuous monitoring and runtime verification. First, unlike traditional computer controlled cyber-physical systems, these systems are mobile and typically operate outside the line of sight of mission operators. The mission operators need to rely on telemetry and sensor data that the systems report remotely, even though the operators may not be computer or systems experts. Second, missions involving UAVs are usually conducted in contested spaces, where these systems are highly likely to encounter adversarial activities. In this project, we considered sophisticated

cyber attacks that attempt to take control of the system. Physical attacks and EM attacks such as jamming were not considered (Section 3.2). Nevertheless, without a continuous signal during the mission that the system is performing as expected, especially when under cyber attacks, it is very hard to claim trusted mission operation. Third, even without cyber attacks, relying solely on pre-mission and pre-deployment tests and analyses to establish that the system will perform in a trustworthy manner is risky. Our ability to formally prove properties of software and create software that is correct by construction, while promising (Section 3.6) is limited. Uncrewed systems involve multiple fairly complex subsystems (such as the flight controller, the payload sensors, information dissemination and retrieval systems) that work in a collaborative manner reacting to external and internal events. Not all components are tested equally, and some of these components may have questionable origin and provenance. Therefore, there is considerable risk in entrusting critical data and operations on these systems.

While mission operators may not understand the internal states and workflows of the various computations which can be numerous as well as complex, they typically have a good idea about the conditions or situations that should not happen in a successful mission. Consequently, trusted mission operation with uncrewed systems requires a meaningful way to reinforce to the mission operators not only the absence of such undesired conditions but also the integrity of such reports, including faithfully reporting the true occurrence of such events. This serves as our final motivation for continuous monitoring and runtime verification. In the context of a trusted and resilient system, a *violation of trust* — when continuous monitoring does report an event or a condition that indicates something that is not expected to arise in a mission — can be used to trigger resiliency techniques such as recovery and automated repair (Section 3.5).

We present the continuous monitoring and runtime verification component of the overall TRMO integrated architecture. This component leverages the *Continuous and Measurable Trust* (CMT) technology developed by BBN under AFRL sponsorship.

3.4.1 Overall Approach and Design Goals

Monitoring and runtime verification comes with its own technical challenges. In particular, we consider the overhead of monitoring, the determination of a baseline model, and the integrity of monitoring.

First, depending on what is being monitored, where, and how, monitoring can slow down the computation being monitored. For instance, if monitoring is performed by a scheduled task, that very monitoring task will now compete for CPU time with other tasks. Monitoring a process's state may require pausing the process and copying the memory. If the probes that collect data from the system are also responsible for analyzing or sending the probed data, the monitored computation process or thread can be blocked non-deterministically. In the context of a flight controller or other onboard computation, large or unpredictable delays introduced by monitoring can disrupt the timing characteristics of the system's operation. For the controller, where timely response is crucial for maintaining a stable flight, this may lead the vehicle to erroneous paths. Delay in disseminating a crucial sensed information may be the difference between mission success and failure. Therefore, one of our key design goal was to minimize the delay introduced by monitoring (i.e., on the monitored processes). We achieved this goal by monitoring only what is needed for runtime verification, returning the control to the main computation as soon as the monitored data is collected, reusing what the systems are already monitoring and reporting (such as telemetry data),

and leveraging efficient support for instrumentation, snap-shotting and tracing that are available in modern OSes such as Linux.

Second, runtime verification assumes that there is a baseline model of some sort, possibly expressed as invariants, that can be checked at run time. These models, embodied as checkable mission-specific constraints or thresholds, are described in CMT as the mission's *trust policy*, and correspond to a multi-dimensional operating envelope that the mission operators can trust. While the operating envelope that the mission operators can describe, understand, and trust is at a mission-level (e.g., a description of how the vehicle should travel in space or how it should behave if the battery runs low) the constraints and thresholds checked by runtime verification are defined in terms of measurements and observations that can be collected from the system. These constraints and thresholds are learned from data collected from sample runs of the mission (including real runs, when possible, and simulations with hardware and software in the loop) with heavy instrumentation and large numbers of probes watching the ongoing mission operation and supporting computation. For instance, data from the sample runs may result in learned constraints such that "in successful missions the vehicle exits waypoint 1 at an altitude of approximately h meters, heading approximately d degrees north, with speed of approximately s meters/hr, and has y percentage of battery left". Policies might also describe source program properties, such as "in successful missions, there is no control or data flow between two subsystems $S1$ and $S2$", or "only process P may access the folder F that contains mission critical data". Such constraints and invariants may be *soft* — sample runs used in training may not be sufficiently diverse and the real mission environment may encounter unexpected situations that may impact flight (such as high wind) or impact battery usage (such as cold temperatures). Therefore, the verification of these constraints at runtime needs to compensate for such differences. We typically addressed this tension by introducing a statistical range to check against instead of concrete values for constraints on physical behavior such as flight paths. For constraints and invariants on system execution and I/O behavior we take a more nuanced approach, and start with a threat model that the mission operators care about. We then consider the software architecture of the system and its dataflow model. Together this approach narrows the focus of what we collect during training runs to only critical components and paths that are most susceptible to attacks that the mission operators worry about and are critical to the success of the mission.

Finally, if monitoring mechanism and the monitored computation are not independent, a successful attack on the monitored component may impact the monitoring mechanism as well. This may cause loss of visibility at best, and may corrupt the monitored data and mislead the runtime verification mechanism at worst. Therefore, keeping monitoring and verification capabilities sufficiently separate and independent from the monitored computation was another key design goal. We addressed this as follows. First, the absence of monitoring is treated as a policy violation, raising an alarm that the system operation can no longer be trusted. Second, we situate the monitoring mechanism in a lower layer in the software (e.g., using a kernel probe), making it harder for an outside attacker to corrupt the monitoring mechanism. We also monitor key pathways for a user-level process to interact with the privileged monitoring mechanism, giving the monitoring mechanism a fighting chance to detect attempts to corrupt it. We also supported hardware-level debugging mechanisms to make our monitoring probes independent of the monitored software. In this case, the component that performs the runtime verification is an independent process separate from the monitoring probes and is further isolated by execution on different hardware.

3.4.2 Monitoring Summary

In summary, we want to minimize monitoring overhead and monitor what is most important to the given mission. Monitoring should also be as independent as possible and must allow for soft verification. As such, what we monitor depends what we want to verify at runtime. We determine what we want to verify by first modeling the behavior in training mission using real flight data and/or hardware/software in the loop simulations (HIL/SITL), and what the mission operators are concerned about (i.e., the threat model). We then organize what we monitor into three main categories: (1) flight behavior; (2) execution of the controller and on-board processes; and (3) I/O behavior (sensor/information management). Collectively these cover a wide range of what a mission operator may care about — including whether the vehicle is flying as planned, and execution integrity of the controller and information management/sensor systems. Our modeling process results in what we call *mission-specific trust policies*. These are constraints and thresholds that can be checked at run time based on data from continuous monitoring. We use telemetry data for the flight behavior, instrument probes in the flight stack and the kernel running the various on-board computational processes to assert integrity of the flight stack's command and control (C2) and I/O.

Over the course of this multi-year activity, CMT expanded beyond its initial support of Pixhawk quadcopter hardware (STM32 M3 32-bit ARM), ArduPilot, and the MAVLink C2 and telemetry protocol. By the end of the activity, CMT had been extended to support ArduPilot running on an Intel Atom-based single-board computers (Intel Up boards using an IA-32 and x86-64 instruction set) and ultimately to support ArduPilot on a Raspberry Pi 3 (which supports 64-bit, 32-bit, and 16-bit thumb-mode ARM). We also incorporated monitoring of a Robot Operating System (ROS)-based network of applications to support inter-component coordination. While CMT's telemetry monitoring is agnostic to architecture, process and task instrumentation may be dependent on architectural nuance (e.g., IA-32 versus 32-bit ARM). In the following subsections we provide details describing monitoring and verification support for ArduPilot and general purpose Linux applications like the ROS-nodes considered in TRMO evaluations.

3.4.3 Monitoring and Verification Support for ArduPilot

ArduPilot is a popular and feature-rich open-source flight control system. To monitor the ArduPilot flight controller on a Raspberry Pi (Pi3) or the Intel Up (Up) board, we applied and enhanced concepts and artifacts developed under the CMT project. Unlike our earlier work, which focused on the all-in-one light-weight Pixhawk hardware running the light-weight Nuttx OS, this effort relied upon much more capable embedded platforms to support ArduPilot. Key advances over our earlier work include the use of hardware with multi-core embedded processors and memory management units that run real-time Linux, as well as external sensor boards to provide bus access to essential flight sensors, such as inertial measurement units (IMUs), magnetometers, and GPS. Collectively, the controller board plus the sensor board provide the inputs and outputs necessary to support GPS-guided autonomous navigation.

Given these two new platforms, we focused our continuous monitoring and verification efforts on the ArduPilot user-space process running on the Pi3 or Up processors. We assume that related runtime verification techniques may be used to assess the trust of Linux, the plethora of supporting software and services, and firmware that support these autopilot devices. Finally, the approaches

outlined below are independent of instruction set architecture of the devices. In an effort to reduce complexity, we aimed to keep CMT dependent only upon user-space APIs and instrumentation strategies which are portable to a wide degree.

To monitor and verify trust, we leverage and apply three software artifacts. First, we developed a Java-based modeling tool for performing ArduPilot and MAVLink protocol modeling tool for defining mission flight (or locomotion) constraints. These constraints statistically describe what a flight plan should look like for a provided mission. Second, we developed a Java-based runtime process for verifying the correctness of autopilot telemetry in-mission. Building upon the modeled constraints of the first tool, this component consumes live device telemetry and performs runtime cross-checking to assess trust of mission tasking and locomotion. Third, we applied select process instrumentation for verification of the correctness of critical aspects of the autopilot's software execution. This type of instrumentation applies a mix of user-space software changes, monitoring points, and kernel-space probes to monitor the ArduPilot task execution for signs of desired and undesired behaviors and states.

Having described CMT's general approach for modeling and runtime monitoring in the previous section, we now provide specific details of the design and implementation, and compare and contrast how the designs vary across autopilot devices (i.e., the Pixhawk, Intel Up, and Raspberry Pi3).

Table 2: Monitoring points for three autopilot devices.

	Raspberry Pi3 (A72) RT Linux OS	Intel Up (Atom) RT Linux OS	Pixhawk (M3) Nuttx OS
Locomotion Constraints	MAVLink telemetry	MAVLink telemetry	MAVLink telemetry
Autopilot Process Constraints	- Wait4 system call	- Wait4 system call	- LR register hook - Image snapshots - Embedded temporal assertions

Table 2 summarizes the data points and sources that support verification of a small uncrewed vehicle's locomotion constraints (flight patterns or ground navigation) and the execution of the software autopilot. The first row describes the data sources used for verifying flight constraints and the second row lists data sources for verifying trust in the ArduPilot process. In the second and third columns we highlight new work under the TRMO program. Prior work under the CMT project is listed in the fourth column for comparison; we provide a brief description here for completeness of this technical report. We now provide a summary of these data sources, methods, and developments for modeling and verifying trusted operation.

3.4.4 Device Locomotion Monitoring and Verification

Across all platforms, we applied and enhanced a common codebase for modeling and verifying flight constraints. In an effort to minimize runtime monitoring costs, we proposed to leverage the MAVLink telemetry as a viewpoint into the internal state of an ArduPilot device. The *MAVLink protocol* is a lightweight and compact command and control and telemetry protocol where each message fits in 263 bytes. It is also used by all of the ArduPilot sub-projects, including ground

rovers or quadcopters developed under this project, to provide a steady stream of internal device states to a mission operator. This makes it an ideal starting point for developing verification strategies and techniques.

We first developed a Java-based tool to support locomotion modeling using the telemetry produced by the AMPRover2 sub-project of ArduPilot. AMPRover2 is the codebase that powers all ArduPilot ground rover devices. CMT's modeling tool ingests a MAVLink stream from a preplanned mission from either simulations (software or hardware-based) or live mission flights on real hardware, such as the Pixhawk. While the stream may contain many types of messages, the modeling phase only considers five unique MAVLink messages types:

1. HEARTBEAT — for tracking aliveness and vehicle type.
2. MISSION_CURRENT — for defining sequenced mission elements (waypoints).
3. SYS_STATUS — for general status messages and battery levels.
4. VFR_HUD — for throttle and ground speed measurements.
5. GLOBAL_POSITION_INT — for latitude, longitude, altitude, and heading.

We further extended the modeling tool to support the quadcopter under the ArduPilot project.

Using attributes and values from these five message types and telemetry data from multiple executions of a given mission as input, we have defined statistic procedures for deriving constraints that define the navigation to each waypoint of an ArduPilot mission. A *waypoint* in ArduPilot/-MAVLink is defined as a marker in Euclidean space. Navigation of an autonomous mission is then defined as a traversal of a set of monotonically-increasing sequence numbers (waypoints) that describe a desired path for driving or flight. Once the modeling process is completed, a set of eight constraints can be cross-checked quickly at runtime against expected values to verify the flight path or ground path is similar to the training set.

In Table 3 we further list each of the eight constraints produced by CMT. This table also details what is measured for the constraint, and then gives the type of assertion that can be produced if a constraint is broken and the potential implication for a given mission. While not all of the constraints were used in the in support of the Red Team testing, the set shown in Table 3 covers both location-based and non-location based checks. Both types of checks can aid verification in the presence of cyber-attacks.

While the modeling portion of the this work is an offline process, the runtime verification of locomotion constraints needed to be integrated into the overall TRMO architecture. As part of our effort, we developed new adaptors and modifications of the CMT runtime constraint checker to: (1) support Arducopter in addition to AMPRover2; (2) run in a headless without a graphical user interface; (3) run locally on the embedded platform of the Up and Pi3; (4) support both TCP and UDP MAVLink streams; (5) generate signals and identifying information about failures for the rest of the TRMO framework; and (6) to be embedded alongside a MAVROS-based navigation control plane. The first five tasks were logical extensions of the pre-existing codebase. The sixth task required redesigning how CMT's runtime component received waypoint sequence information. Unlike our prior work, where the information was fully encapsulated in a MAVLink telemetry stream, in this new configuration a ROS node is responsible for publishing waypoint tasking directly to the ArduPilot binary. As a result, the ArduPilot telemetry stream would not contain the

information necessary for locomotion verification and we were required to extend CMT to encap-sulate and decode these MAVROS signals to ensure locomotion verification.

Table 3: Summary of waypoint window constraints.

"Normal" Starting Position for Waypoint Window (is localization dependent)	
Measured	Starting position for window (position estimate and heading)
Assertion	The MAV's starting position and orientation is different from the modeled case.
Implication	The MAV is approaching navigating to the next WP from a new initial position.
Implication	The supporting model might be off or incomplete for the real world.
Vehicle's First Alignment to Next Waypoint (is localization dependent)	
Measured	Both the position and time taken to align the MAV to the next WP
Assertion	The MAV having a hard time adjusting/aligning to the next WP.
Implication	Possibly due to environment, starting position, configuration or interference.
Implication	If the MAV is off course, the supporting model might be off or incomplete.
Realignment Operations After First Alignment (is localization dependent)	
Measured	Once MAV is aligned, what does course adjustment look like (heading or CoG)
Assertion	The MAV is oversteering.
Implication	Struggling to navigate, possibly due to environment, configuration, or interference.
Distance Traveled (may be localization dependent. depending on device telemetry)	
Measured	Average distanced traveled to reach the next WP (e.g., sum of GPS readings)
Assertion	The MAV traveling farther or less than normal.
Implication	The MAV may not have resources to complete navigation; model might be off.
Implication	The MAV might get to the next WP sooner/later, thus missing a timed deadline.
Time Taken	
Measured	Average time to navigate to the next WP. (time for WP window traversal)
Assertion	The WP traversal is taking longer than normal.
Implication	The MAV may not reach the next WP in a timely manner. Miss deadline.
Resources Remaining	
Measured	Battery usage for this WP window. (i.e., battery voltage over cells)
Assertion	The MAV using more power than normal. (less usage should not be an issue)
Implication	The MAV might not have enough resources to reach the next WP.
Implication	The MAV may not have enough resources to complete the mission.
Ground Speed	
Measured	Groundspeed for the MAV (e.g., VFR groundspeed)
Assertion	The MAV is moving too fast or slow.
Implication	The MAV might reach a destination sooner or later than required.
Implication	The MAV may miss a mission deadline.
Throttle	
Measured	Throttle for the MAV (VFR Throttle)
Assertion	The MAV is at an undesired set point.
Implication	The MAV might reach a destination sooner or later than required.
Implication	Could be a response to environmental conditions.

3.4.5 Autopilot Task Monitoring and Verification

For this evaluation, we focused on monitoring the Autopilot for liveliness. While there are many approaches to asserting liveliness, we choose to use a fail-fast failure detection technique. For ArduPilot, CMT's constraint checker can partially assert this style of liveliness by detecting the absence of telemetry. However, such an approach in itself is not sufficient, as it does not account for network delays and other interprocess communication effects. To supplement that style of detection, we developed a Python- and Linux-based watchdog parent process for managing the ArduPilot task. On task creation, the user-space watchdog would enter a wait state for OS signals regarding the child process. Upon child process exit, the watchdog would signal the TRMO system. This approach is suitable for monitoring real-time processes, like ArduPilot, as it is light-weight, blocking, and in-line with the task execution. As a result, it incurs little overhead and produces failure notifications quickly.

While TRMO's requirements for verification of the ArduPilot task were straight forward, we also examined both the task integrity and availability of the ArduPilot process on Pixhawk (see Table 2 for the monitoring points). Unlike the Pi3 and Up boards, there are several differences on the Pixhawk. First, the ARM Cortex M3 is a light-weight micro-controller and only has a memory control unit to aid the Nuttx operation system. As such, it does not have a basis to enforce fine-grained memory permissions. Furthermore, all tasks on the Pixhawk execute with the same privileges. While these facts should be perceived as inherent risks, they are justified by the typical use of the device — it is a specialized cyber-physical system for autonomous flight control only and it only has a limited control channel.

Building on top of Pixhawk, ArduPilot, and Nuttx OS, we developed a set of specialized run-time instrumentations including: a (1) Nuttx API monitor by sampling the LR register on the ARM device; (2) task-, process-, and hardware debugger-based snapshot analysis techniques to perform state integrity checking; and (3) embedded temporal logic assertions for the ArduPilot process to guard access to sensitive states like waypoint tasking and device parameters. We applied similar verification concepts to examine the integrity of general purpose Linux processes on the Raspberry Pi3. We describe those methods in the next section.

3.4.6 Continuous Monitoring Support for Linux Processes

Unlike the dedicated Pixhawk autopilot device, TRMO also supports powerful embedded devices and new applications for MAVROS-based flight control, such as an image server for exposing camera sensor data to end-users (Section 4). At the outset, the addition of the Up and Pi3 boards allowed us to use Linux, as opposed to the light-weight Nuttx OS, and to support additional applications. The first such rich end-user application provided a more systematic integration and control plane over the ArduPilot software and allowed us to connect image sensors and tasks, such as an OpenCV color-processing node. The second rich end-user application included new functionality to support camera-based mission operations.

Table 4: System call classes and examples for runtime monitoring

Class	Examples	Description
User or group ID changes	sys_*[u—g]id	To track user or group ID change for process. Will alter the set of resources that may be accessed.
Block storage access	sys_mount, sys_fsopen, sys_fsmount	Usually privileged instructions. Used to change the set of file system resources available.
FS access and manipulation	sys_openat, sys_open, sys_dup, sys_*own	To track how file system objects are accessed.
Process and capability management	sys_*cap, sys_exit, sys_*kill, sys_clone, sys_execve	To track how new and existing processes are changed and related.
Memory image changes	sys_brk, sys_mmap*, sys_macl	To track memory layout, loading/unloaded, and permissions.
IPC and network communications	sys_splice, sys_pipe2_sys_mqopen, sys_shm_open, sys_connect, sys_accept, sys_send*, sys_recv*	To track data flows to-and-from a process.

Since these applications are user-space processes executing on Linux, we were able to explore general-purpose monitoring support on Linux. As such, we experimented and developed capabilities on top of the Linux kernel's Audit and Extended Berkley Packet Filters (eBPF) trace APIs, and using the `inotify` APIs of Linux. Using these APIs, we developed user-space monitoring processes that registered a privilaged interset in a collection of low-level events, including operations such as creating new proccesses or accessing file system resources. In Table 4 we summarize the class and collection of events that we were able to monitor using these APIs. These probes allow us to monitor and verify execution behaviors against training sets recorded while profiling the applications, and then to signal violations of trust as low-level events and their attributes deviate from expected profiles (e.g., white- and black-list violations of system call types or arguments).

3.4.7 Summary of Continuous Monitoring and Verification

Under the Trusted and Resilient Software-intenstive Systems (TRSYS) technical area of the Trusted Computational Information Systems program, the CMT project, and the TRMO project, we have developed methods and tool support to perform continuous monitoring and verification of trust in an small unmanned vehicles. We have applied this approach to assess trusted mission operation over three autonomous autopilot platforms and their supporting general purpose Linux processes, such as image servers and ROS-controller nodes. Using this approach, operators can gain a degree of trust in a mission support system's locomotion characteristics and on-board task execution. As a result, they can receive alerts when the device is not behaving as expected, thus signifying a degraded trust level. We integrated and applied these monitoring and verification capabilities into a joint system providing a degree of last resort fight-through cyber capability. In this integrated system and subsequent evaluation, continuous monitoring and verification played a key role in triggering resilient repair operators.

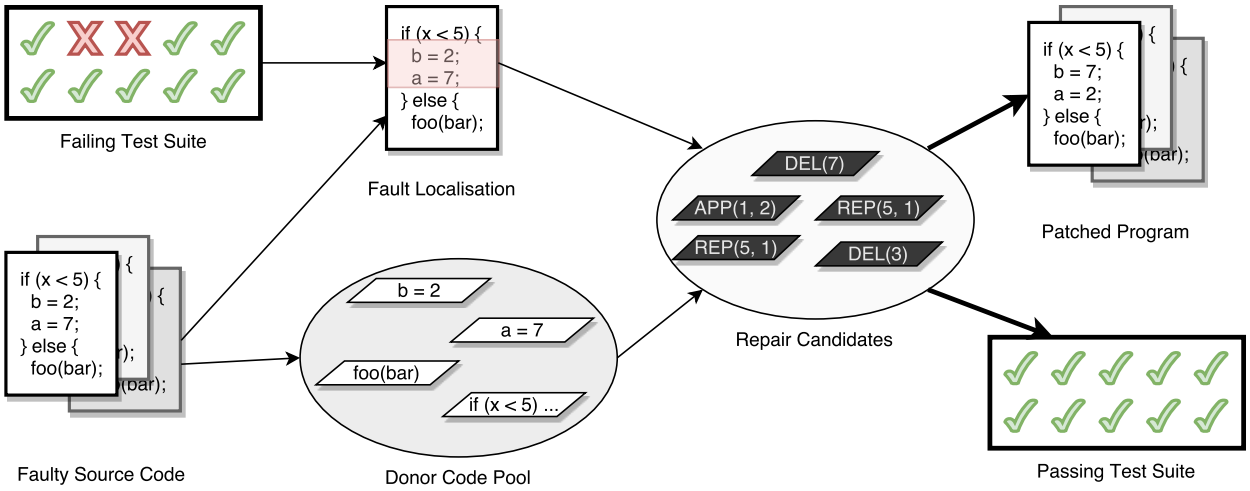

Figure 7: A high-level overview of the search-based program repair used by Darjeeling. (Adapted from [78].)

3.5 Repair Component

In response to an attack on the UAV controller, TRMO invokes a framework for search-based program repair to automatically repair the associated vulnerability in the controller source code. An *automated program repair* algorithm takes as input a buggy program and evidence of correct and incorrect behavior and produces, automatically or with minimal guidance, a patch to address the buggy behavior [43, 48, 50, 51, 54, 65, 81, 83]. To this end, we adapt *Darjeeling*,[5] a language-agnostic framework for search-based program repair, to operate effectively in the embedded system context. This approach operates on unmodified source code, without a need for special coding practices or annotation.

In the context of an autonomous vehicle, automated program repair must especially prioritize *safety* and *efficiency*. Safety is required to ensure that the execution of arbitrary code introduced by candidate patches does not interfere with the safe operation of the vehicle. Efficiency is required to ensure that an acceptable repair is found in a timely manner, before battery is depleted or the mission fails. Furthermore, the repair process must run on an embedded platform with minimal resources and no internet connection.

Section 3.5.1 provides relevant background on search-based program repair. Subsequent sections elaborate on repair activities, including include key innovations required for successful, efficient repair in the embedded context.

3.5.1 Search-based Program Repair

Search-based program repair transforms the problem of repairing software bugs, represented as failing test cases, into a search for a patch to the program that leads it to pass the previously failing

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

tests (i.e., fixes the bug) while preserving the correct functionality of the program, represented by the passing test cases [52]. Patches typically take the form of source-level changes, but may also be represented by binary-level modifications [71].

The majority of search-based repair techniques exhibit the same high-level structure, illustrated in Figure 7. Broadly speaking, the repair process consists of three steps: *fault localisation*, *patch generation*, and *search*.

Fault Localization. The first step of the repair process is to determine the likely location(s) of the bug(s) via a process of *fault localization*. Fault localization assigns a suspiciousness values to each modifiable unit within the program (e.g., a statement or source code line), measuring the likelihood that the unit is (partly) responsible for the bug [36]. Most search-based repair techniques use a lightweight technique known as *spectrum-based fault localisation* (SBFL) to compute suspiciousness values based on test suite coverage [62, 68]. SBFL first computes coverage information for the faulty program to determine the set of components (e.g., lines or statements) that are executed by each test. A suspicious value is then calculated for each component based on the number of passing and failing tests that execute (or cover) it. In general, components that are more frequently associated with test failure are assigned a higher suspiciousness value than those that are not; components that are never executed by any of the failing tests are assigned a suspiciousness value of 0, excluding them consideration altogether.

Suspiciousness values are used during the search to focus the generation of candidate patches towards the areas of the program that are deemed most likely to be faulty.

Candidate Patch Generation. Given a set of suspicious program components, a program repair technique then constructs a set of candidate transformations (or *edits*) for each component. Collectively, the set of candidate program transformations across all of the suspicious components within the program is known as the *edit space* or *fix space* [49].

Program transformations are generated using transformation schemas (also known as *repair operators* or *repair templates*) and a pool of donor code snippets [44]. Each transformation schema describes a template for a program transformation, and may contain a number of holes (i.e., free variables) that are filled using snippets provided by the donor code pool. A number of different repair operators have been proposed in the literature. For example, the GenProg, AE, and RSRepair techniques share a common set of *statement-level* transformation schemas, intended to be generic enough to fix arbitrary bugs: Statements may be deleted, swapped, replaced, or appended. Existing statements from the same file are used as ingredients during statement replacement and appending; this principle exploits the *plastic surgery hypothesis* [9], which states that many bug fixes can be constructed using existing code from the buggy program (an idea that underlies many repair techniques [43, 67, 82, 84]). Other approaches use templates of finer granularity admitting, e.g., sub-expression manipulation or other more complex repair templates intended to address specific bug classes [40] (e.g., by adding a missing null check).

Search Algorithm. Given the above-defined search space, the final challenge concerns the efficient sampling or traversal of this space. This is commonly achieved using heuristic [82] or meta-heuristic search. In most cases, candidate patches consist of a single edit [41, 48, 67, 82, 84],

but a small number of techniques also permit multi-edit patches [46, 78]. For example, the GenProg technique [43] uses a genetic programming heuristic to search for patches, represented as a sequence of edits, via a continual process of mutation, crossover, and selection (with fitness defined by the number of test cases a patched program passes). RSRepair [67] and AE [82] operate using the same edit space as GenProg, but employ different search algorithms: RSRepair uses random search to sample the space of single-edit patches, while AE usies static analysis to weakly identify and prune semantically equivalent patches and heuristically (but deterministically) evaluate them.

3.5.2 Embedded System Repair Overview

At a high level, Darjeeling accomplishes safe and efficient embedded repair. Figure 8 outlines this automated repair process.

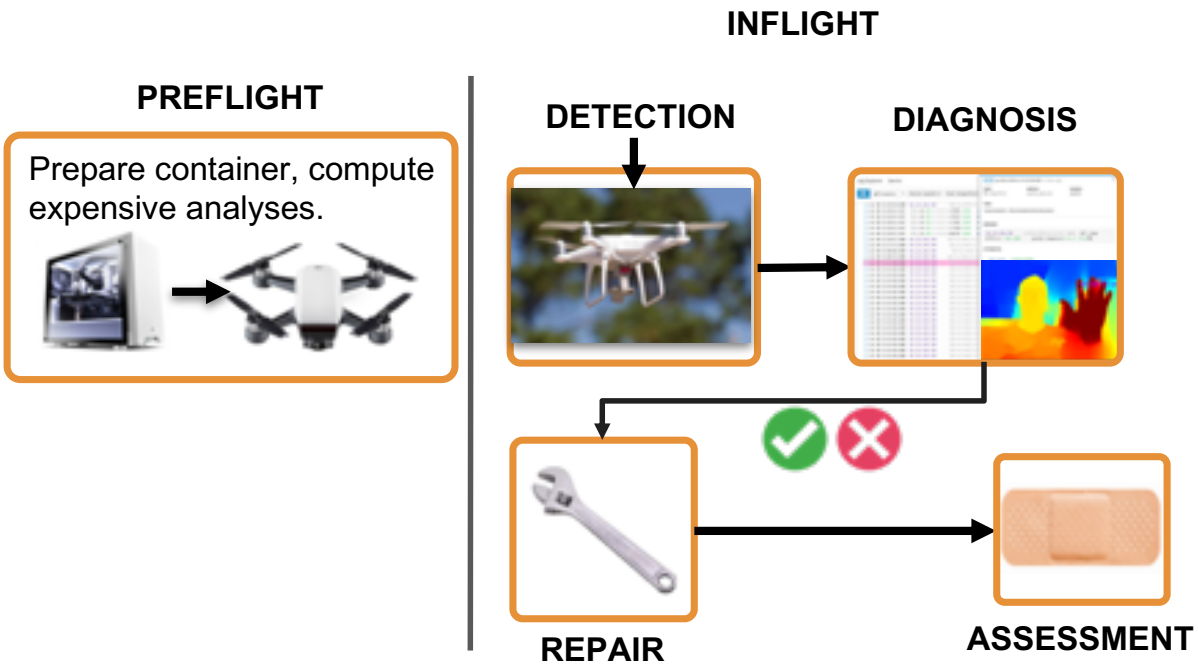

Figure 8: A high-level overview of the Darjeeling automatic repair process in TRMO, shown for an aerial vehicle.

The automated repair process involves the following steps:

1. **Pre-mission computation.** Darjeeling uses a combined online/offline architecture, conducting significant pre-mission computation of expensive analyses to support efficiency. That is, Darjeeling lifts key elements of the repair process, including expensive test suite coverage analysis, system containerization, and set of static analyses [82], to an *offline* preprocessing step. Only information directly related to the attack or vulnerability itself — which is unknown pre-mission — is computed *online*. The offline stage occurs pre-mission (i.e., before the vehicle leaves to complete its mission); it need only be (partially) repeated when the vehicle source code or configuration are changed.

2. **Detection.** The online stage of the repair process takes place onboard the vehicle and begins when either the CMT or Zipr component of TRMO informs the Darjeeling repair component of an attack.

3. **Diagnosis.** Darjeeling uses the information provided by the detection tools and then further integrates with system monitoring, mission logs, and the previous sequence of commands executed by the vehicle, to construct executable test cases that safely recreates the attack in simulation.

4. **Repair.** The repair search process involves several fundamental innovations to support both efficiency and safety. (1) Significant program analysis, leveraging precomputed coverage and static information from the pre-flight phase, determines the source code lines executed during the attack and computes suspiciousness and efficiently generate a set of candidate source-level patches. (2) Containerization, low-fidelity simulation, significant simulation speedup, and careful test suite prioritization allow for efficient evaluation of candidate patches, allowing Darjeeling to exhaustively search for a plausible repair that allows the vehicle to pass both the pre-existing test suite and the recreated attack.

5. **Additional assessment.** In simulation, Darjeeling uses inexpensive behavioral indicators (e.g., log messages) to determine whether the modified source code will lead the embedded system to behavior correctly. Upon discovering a plausible repair using these inexpensive oracles, Darjeeling forwards that repair to CMT for more extensive trust evaluation while it continues to search for other plausible repairs.

The Repair/Assessment search continues until a resource limit is reached (e.g., wall-clock time) or the CMT component instructs Darjeeling that a plausible patch has passed its trust evaluation. We provide additional detail on these components subsequently.

3.5.3 Pre-mission Repair Computation

Pre-mission, the Ground Control System (GCS) prepares the system for the mission. This step *lifts* expensive portions of the repair process to an offline preprocessing step. The GCS may be computed on virtually any available hardware, including commodity, consumer-grade laptop or desktop computers.

The precomputation phase addresses three concerns:

- **Containerization.** Darjeeling uses commodity containerization technology, and specifically Docker,[6] to safely and efficiently evaluate candidate repairs (Section 3.5.4). During the precomputation phase, TRMO builds a Docker image for the autopilot software system.

- **Static analysis.** We feed the constructed Docker image to Kaskara,[7] a language-agnostic static analysis platform that we developed, to scan the source code and to identify the set of all statements within the program via a static analysis. Specifically, Kaskara uses a language-specific plugin to identify the statements within the program for a particular language, and

[6] http://www.docker.com
[7] http://github.com/ChrisTimperley/Kaskara

to extract additional semantic information about those statements (e.g., variables in scope, liveness, relevant types) that is later used during the online phase of the repair to reduce the size of the search space by weakly identifying equivalent repairs. Kaskara uses the well-established Clang static analyzer as its C++ plugin. Note that this static analysis step is relatively expensive in terms of memory and compute resources (i.e., wall-clock time), and either takes a prohibitively long time (approximately 15 minutes) to run on the embedded device, or else exceeds the available memory. Running this step on the GCS takes approximately 2-to-3 minutes on a consumer-grade laptop and avoids introducing a sizeable overhead to the repair process.

- **Coverage collection.** Using the Docker image built during the precomputation phase, we add lightweight instrumentation to the program and collect coverage information for a system-level regression test suite, composed of simulated missions. The generated coverage information is saved to disk on the embedded device and is later used to improve the accuracy of the fault localization at the beginning of the online phase of the repair process.

3.5.4 In-mission Repair Computation

Fault Localization. As part of our efforts to reduce the expected cost of finding an acceptable patch, we use *spectrum-based fault localization* to determine the most suspicious areas of the program. During the repair process, we restrict the generation of candidate patches to those areas of the program that are deemed most suspicious.

Repair Selection. We investigated methods for improving our ability to repair complex bugs, such as those requiring two or more edits [70]. The most promising is the selection operator, which chooses which candidate repairs to investigate. We integrated *lexicase selection* [34] into our tools and compared its performance to the current selection method (tournament selection).

Lexicase selection emphasizes performance on individual test cases rather than aggregating fitness across all available test cases. This encourages a population of candidate solutions to preserve diversity and explore the search space more effectively. It is particularly effective for *modality* problems, where a system must respond in different modes when given different inputs. For automated repair, the different modes correspond to the program's execution on different inputs. While the use of lexicase selection for automated program repair is not complete, our prototype implementation found repairs more frequently (higher success rates across multiple random restarts) than seven other selection algorithms, including the state-of-the-art tournament selection approach.

In brief, the lexicase algorithm works as follows: (1) place test cases in random order; (2) iterate through the tests according to (1) and on each iteration eliminate all individuals that fail that test; retain the remaining individuals; repeat steps (1) and (2) until a fixed-sized population of individuals have been generated.

The quality and orthogonality of the test suite affect the effectiveness of lexicase selection compared to other methods. In particular, the uniqueness of statements executed by each test case is positively correlated with the effectiveness of lexicase selection. The more diverse a test suite is, the more likely it is that emphasizing performance on individual will improve performance.

Repair Generation. For TRMO, Darjeeling used the Rooibos[8] library to support a rich and customizable set of repair templates for arbitrary languages, including C++.

The templates used induce the space of program transformations from which candidate patches should be composed. We first consider three "classical" statement-based transformation schemas based on those introduced by GenProg [83]: delete-statement, replace-statement, and prepend-statement. In addition, we make use of domain-specific repair templates (cf. [40]).

Search Space Optimizations. We consider a number of optimizations to reduce the search space of candidate transformations considered. In general, assessing the correctness of a candidate patch in simulation (i.e., running a simulated mission using modified flight control software) is the most expensive aspect of automated program repair in TRMO. As a result, any precalculation, logical filter, or effective heuristic that can rule out certain candidate edits as unlikely to lead to fruitful repairs is critical. In TRMO, search space optimizations were crucial for conducting repairs in a 5–10 minute live mission setting.

Optmizations used, developed and refined as part of TRMO to filter the space of repair transformations include:

1. **use-scope-checking:** ensures that all variable and function references that occur in a given transformation are visible from the scope into which they are being inserted.
2. **use-syntax-scope-checking:** ensures that any keywords introduced by a transformation (e.g., break and continue) are permitted by their surrounding context.
3. **ignore-dead-code:** prevents the insertion of code that exclusively writes to dead variables.
4. **ignore-equivalent-prepends:** uses an approach inspired by instruction scheduling to prevent equivalent insertions of code.
5. **ignore-untyped-returns:** prevents insertion of a return statement into a context where the type of the retval is incompatible with the return type of the enclosing method or function.
6. **ignore-string-equivalent-snippets:** transforms donor code snippets into their canonical form, thus preventing the insertion of string-equivalent snippets.
7. **ignore-decls:** prevents transformations that are either applied to declaration statements, or else solely introduce a declaration statement.
8. **only-insert-executed-code:** prevents the insertion of code that has not been executed by at least one test case.

3.5.5 In-mission Repair Evaluation

We desire repairs that retain required semantics while mitigating the bug or security vulnerability. Operator trust in software systems is a complicated topic in general (e.g., [13]) and the factors that influence operator trust in automated repairs in particular are still being studied (e.g., [69]). Generally, independent of any concern related to how the repair is presented syntactically (e.g.,

[8] https://github.com/squaresLab/Rooibos

proper indentation, use of comments [26], etc.) we desire that patches be held to the highest available rigorous standard of automated testing.

This results in a natural tension: trustworthy repairs require significant testing which requires long delays, but fighting through a mission to provide resiliency admits only small delays. We address these concerns through two insights. First, we reduce the overhead associated with high-fidelity software simulation of candidate repairs. Second, we make use of a two stage process for repair evaluation.

Low-Overheard and High-Fidelity Repair Simulation. Repairs to control software for autonomous vehicles cannot be tested directly via live locomotion. For example, a low-quality candidate repair, if deployed, might cause an uncrewed aerial vehicle to crash. As a result, we employ high-fidelity simulation to evaluate repairs. In addition, we desire to evaluate multiple candidate repairs in parallel whenever possible to minimize latency [82]. This requires minimize interference between multiple candidate patches and a single set of vehicle hardware.

To prevent such interference with minimal compromise to efficiency, we use BugZoo [79], a container-based platform, to safely evaluate each candidate patch within a sandboxed container. Furthermore, the process of evaluating candidate patches in parallel is simplified by using a separate container for each patch. Darjeeling uses Docker technologies to simplify the parallel evaluation of candidate patches and to shield the host machine from the side effects of executing candidate repair code.

Two-Stage Repair Evaluation. In software testing, the *oracle* formalism is used to describe a correct answer or set of behaviors with respect to an input. We consider two separate oracles when evaluating candidate repairs. First, the regression tests associated with the autonomous vehicle control software typically include indicative missions and high-level notions of correctness. For example, informally, a test might say "starting from this location, if we instruct the flight control software to visit these three waypoints in sequence, it will visit them in sequence and return to the start in a given amount of time".

One advantage of such simple oracles is that they can be evaluated in software simulation using an *accelerated logical time*. That is, a mission that would take ten minutes in the real world might be *simulated* at a certain degree of fidelity and run to completion in one minute. A combination of parallel, containerized execution and accelerated logical time allows TRMO to evaluate multiple candidate patches and missions in much less real-world time than it would take to complete even a single mission. In TRMO, this admits a $40\times$ simulation and assessment speedup.

However, these simple oracles may not be adequate in the face of security vulnerabilities or bugs. For example, flight control software that can be subverted to visit an extra, enemy-controlled location in addition to the specified waypoints may still pass a test that just checks to see if it visited the specified waypoitns. Similarly, whether or not flight control software leaks sensitive information may be checked by such a simple oracle.

In TRMO, the CMT analysis and deployment component is responsible for the runtime assessment of trust (Section 3.4). CMT can determine if the vehicle is subverted to visit an extraneous location or if sensitive files are accessed in an unplanned manner. However, because of its higher standards for correctness, it also incurs a higher overhead. While the overhead is minimal when running a single copy during a mission, it becomes a consideration when we desire to evaluate

multiple candidate patches on multiple missions using an accelerated logical notion of time.

We thus investigated and implemented a two-stage solution in which candidate repairs are initially evaluated in accelerated simulation. Only those candidate repairs that pass that stage are subjected to the more thorough CMT-based assessment of trust. The first step acts as a sieve: the majority of incorrect candidate patches are rejected rapidly.

3.5.6 Darjeeling Repair Summary

The primary goal of the repair component is to provide resiliency by changing control software in the face of security attacks or latent defects. We desire to construct such repairs in a rapid manner (to allow the mission to be continued) but also in a manner that operators can trust. Our Darjeeling algorithm reduces the overhead of searching for a repair by splitting the process into distinct pre-mission and in-mission phases, by offloading as much work as possible to the preflight phase (e.g., expensive static analysis), and by a careful two-stage evaluation of candidate repairs.

3.6 Formal Methods Component

One of the goals of TRMO is to produce trusted techniques that admit human operator confidence in the correct deployment of the system. To that end, we developed and evaluated three formal or algorithmic components. First, we created architectural proofs of an aspect of system correctness: the interplay between the trust monitoring component and the protected software. Second, we devised novel algorithms to reduce the verification and validation costs associated with automatically-generated software repairs. Third, we proved certain properties of our binary hardening and diversification system.

3.6.1 Architectural Proofs

TRMO included an analysis of the resilient systems developed, especially their architectures, and the creation of formal and semi-formal models of them to help increase trust. One resilient architectural pattern is the *dual-controller pattern*, in which a simple, trusted controller monitors a more complex locomotion controller, detects when it has been attacked, takes over control of the vehicle temporarily while repairing the locomotion controller's software, and hands back control after the repair. This is one of the primary methods supported by TRMO's analysis and design component (Section 3.4).

We created a formal model of this dual controller architecture, represented in the language of the ACL2 theorem prover [38]. In particular, we proved a key property of this model: that exactly one controller is controlling the vehicle at any time. We also analyzed other aspects of the systems team and provided explanatory documents, including sequence diagrams modeling the interactions of various components in the systems, to support manual trust of unproved aspects of the system.

Formalization and Proof of Dual-Controller Hand-off. We created a formal model of the dual-controller architecture in the ACL2 theorem prover. In TRMO's dual-controller approach, an untrusted *locomotion controller* runs flight control software and mission-level payloads while a *trusted controller* is responsible for runtime monitoring, trust assessment, and repair. However, sensors and actuators in the autonomous vehicle are not duplicated, so the trusted and locomotion

controllers must share or multiplex access to them. Correctly switching between which controller has access to the sensors and actuators is necessary for trust in the system: for example, once an attack has been detected, the compromised locomotion controller must not be allowed to control the vehicle.

Our approach was to model the locomotion controller and 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 property — that exactly one controller is controlling the system (operating the actuators) at any time — and produced a machine-checked proof of the property using ACL2.

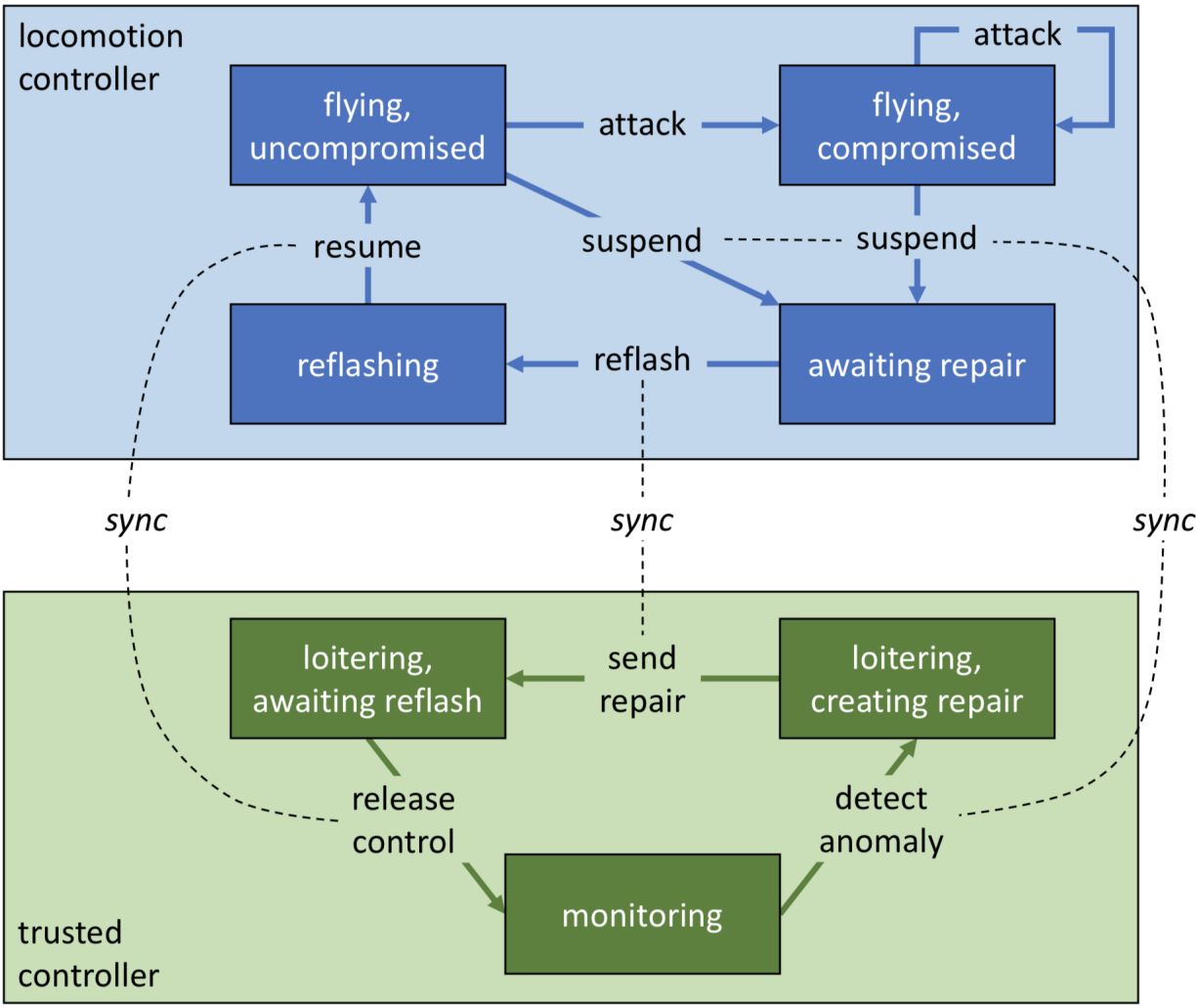

Figure 9: Hand-off protocol for dual-controller architecture. The correct operation of this protocol was formally proved for TRMO.

The state machine is depicted in Figure 9. There are two interacting state machines: one for the locomotion controller, in blue, which controls the vehicle under normal conditions; and

one for the trusted controller, in green, which controls the vehicle under exceptional conditions. The abstracted states of these state machines are indicated by rectangles (four for the locomotion controller, three for the trusted controller); the transitions of these state machines are indicated by solid arrows; the dashed lines link transitions between the two machines that are *synchronized*, i.e., happen at the same time.

During normal conditions, and in particular initially, the locomotion controller is in the 'operating, uncompromised' state, while the trusted controller is in the 'monitoring' state. When an attack happens, the 'attack' transition moves the locomotion controller to the 'operating, compromised' state; further attacks (i.e., self-loop transition 'attack') keep the locomotion controller in that state. When the trusted controller, as part of its monitoring activities, detects an anomaly (due to the locomotion controller being compromised), it takes control of the UAV, via the 'detect anomaly' transition that takes the trusted controller to the 'loitering, creating repair' state; this transition is synchronized with the 'suspend' transition of the locomotion controller (which in principle could take place from an uncompromised state, if the detected anomaly is a false positive; this is why there are two 'suspend' transitions), so the locomotion controller goes to the 'awaiting repair' state. When the trusted controller (or some component communicating with it) has created a repair, the 'send repair' transition takes the trusted controller to the 'loitering, awaiting reflash' state, where it is still in control of the UAV; this transition is synchronized with the 'reflash' transition, which takes the locomotion controller to the 'reflashing' state.[9] When the reflashing is complete, the trusted controller releases control of the UAV via the 'release control' transition, which takes the trusted controller back to the 'monitoring' state; this transition is synchronized with the 'resume' transition, which puts the locomotion controller back in control of the UAV, in the 'operating, uncompromised' state.

Given the state machine model of the system and the statement of the property discussed above as an ACL2 theorem, the actual proof generation and proof checking were highly automatic.

Additional TRMO System Analyses. We also analyzed, in more detail, two versions of the TRMO prototype systems used in the Red Team evaluations, focusing on architectural properties. We created a variety of semi-formal models (e.g., sequence diagrams) of the systems.

For the ground rover demonstration system, we reviewed the TRMO software-in-the-loop harness code. By a process of formal review, including independent questioning of programmers, we identified weaknesses in early TRMO implementations that were fixed before deployment. In addition, we clarified differences between the simulation and the expected hardware system. We also produced three documents clarifying the TRMO system and approach to simulation:

1. a table of file handles and messages
2. a UML sequence diagram of processes involved in the simulation
3. a UML sequence diagram of control flow between modules in the simulation

For the aerial quadcopter demonstration, we again formally evaluated the architecture (the system for aerial vehicles is not identical to the system for ground-based vehicles). We created a

[9] In many autonomous vehicles, such as the ground-based rovers considered by TRMO, locomotion control software is maintained in special non-volatile storage and is updated through a process known as *flashing*. For autonomous vehicles with standard hard drives, flashing corresponds to normal filesystem updates.

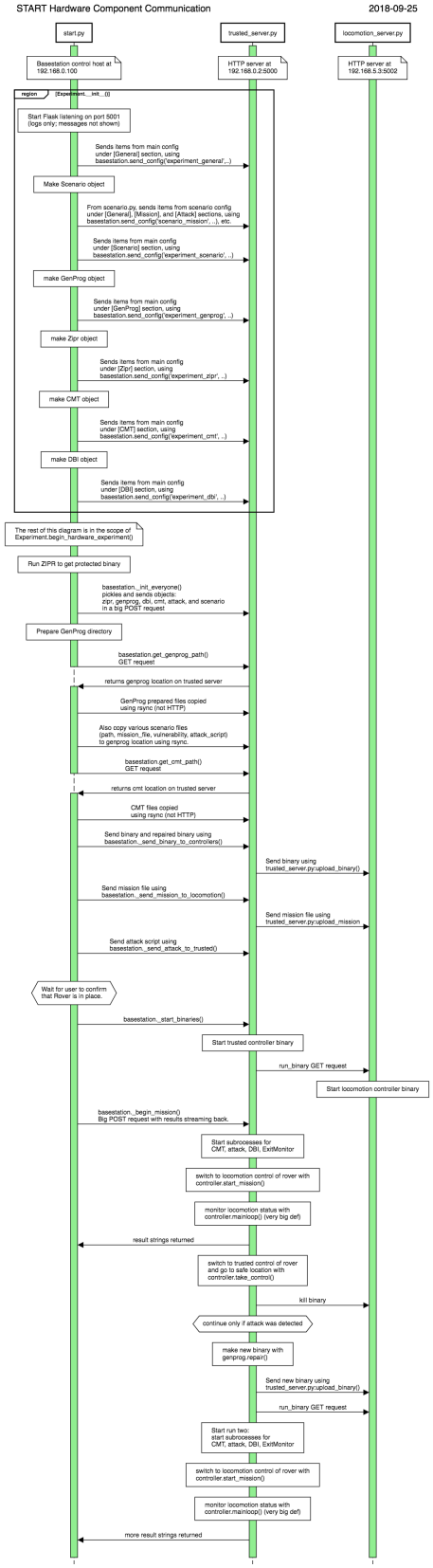

Figure 10: Formal sequence diagram of communication between TRMO and aerial vehicle hardware components.

sequence diagram capturing communication between the TRMO framework and the autonomous vehicle hardware components, shown in Figure 10.

3.6.2 Reducing repair verification and validation costs

Search-based software repair methods, such as Darjeeling and GenProg, propose patches that are correct with respect to a test suite (Section 3.5). That is, a *repair* is defined to be a software patch that passes both the bug-inducing test case(s) and also all other available tests [83]. However, test suites may be *incomplete*, failing to test all required functionality, and the repair may inadvertently break some of this untested functionality [42, 73]. A second concern involves *clever mutations* — repairs that technically pass a test but do so by evading the test's intent (e.g., [80]). Consider a patch for a buggy sorting program that modifies the program to always return an empty list. If the test simply checks that the output is in sorted order, but does not verify that it contains all of the input values, then the patch could be accepted as a repair even though it fails to actually repair the bug. This is an example of a clever mutation that evades test intent; the test's comparison of the given answer against a gold-standard oracle does not match developer intent [10] and thus does not provide operator trust. Addressing these concerns would improve the trustworthiness of search-based repairs, and understanding the effect of a patch is a key element of that trust.

A different threat to patch trustworthiness arises because the patches proposed by search-based repair methods often look quite different from the repair a human would have crafted and require significant human effort to determine if they are correct [73, 77]. Further these methods often generate a multitude of candidate patches that require manual inspection for validation. Since there are often too many candidates which would require too much time to feasibly inspect, we developed a solution to reduce the cost of evaluating automatically-generated patches as produced by search-based methods.

Earlier research proposed highlighting syntactic differences between programs, for example, by providing a `diff` file [43]. Although `diff` notation provides a compact representation of syntactic changes, it can be difficult to infer the changes to program functionality which they represent. Further, programs can have the same functionality but different internal constructions, so it is possible that a single defect could be repaired correctly in multiple ways. Therefore, we chose to focus on semantic program invariants instead of syntactic `diff`s. An invariant is a logical predicate over program state (e.g., program variables) that is true on every correct run of the program. Formal invariants include loop invariants and assertions; they are useful for documenting programs, detecting anomalies, localizing faults, and proving systems correct [8, 25, 55–58].

Our approach incorporates formal methods into the repair process, using them to show that automatically-generated patches preserve certain aspects of required functionality and to highlight the key semantic changes implied by a proposed patch. Conceptually, we achieve this by generating and comparing relevant program invariants for the original program and the proposed repair. This comparison determines if the patched program violates an important known property of the original program, and it identifies invariants that describe the repair's corrected functionality. We thus avoid an impediment to using formal methods in search-based program repair — a buggy program is incorrect in at least some of its behavior, and therefore does not (currently) represent a correct specification.

In addition, we partition (or cluster) patches into semantically equivalent clusters, guiding and reducing manual validation efforts, reducing verification and validation (V&V) costs. The key

insight is that that two patches may have different syntax (e.g., different `diffs`), but if they have provably-equivalent semantics, then they will also have equivalent functional behavior on tests. Developers need only inspect inspect one indicative candidate patch per partition: if that candidate patch is found to be inadequate, for example, then all other patches in that partition are also inadequate.

We use dynamic invariant generation to create a set of invariants, one for the buggy program and one for each proposed repair. Since most software bugs are revealed through a failing input, we use the failing tests, together with the supplied test suite, to generate invariants that are most relevant to the repair. For our autonomous vehicle evaluations, the failing test takes the form of the captured attack input coupled.

If two patches have invariants that logically imply each other then those two patches are functionally equivalent. However, determining logical implication can be expensive for complicated invariants resulting from real-world software. As a result, we also consider approximation. We group repairs into quasi-equivalence classes using one of two distance metrics, one based on logical implication and one based on syntactic comparison of the invariant sets. The syntactic distance metric approximates full logical implication but is less expensive to compute. Each class is made up of invariant sets that are distance zero from each other under the chosen metric. We use hierarchical clustering to highlight how the classes are related.

In addition, we explicitly compare the invariants of each patch to those of the original buggy program. This approach highlights the key semantic differences between the original program and each patch, and it allows a developer to quickly consider multiple possible patches by evaluating only one element from each semantic class.

Our algorithm for reducing the V&V costs associated with automated program repair based on partitioning patches into (quasi-equivalence) classes is called *PatchPart*. PatchPart is an algorithm for understanding automatically-generated patches through symbolic invariant differences. It takes as input a set of programs and computes the pairwise semantic distance between them (in invariant space) and clusters the programs into semantically-similar classes based on those distances. Programs are evaluated using a dynamic invariant generator, a supplied regression test suite, and the bug-inducing tests. This produces a set of inferred invariants for each program. Next, pairwise distances between sets of invariants are computed. The distances are used to create a hierarchical (partial) ordering of clusters, and each program is placed into a cluster based on its location in the hierarchy.

More specifically, PatchPart constructs a set of pre-conditions and post-conditions for each function in each considered program. TRMO uses the well-established Daikon [21] tool to produce dynamically-generated program invariants. PatchPart then calculates the invariant distance between each program pair using one of two methods. We consider two methods for computing the pairwise distance between sets of invariants: formal implication between logical formulae and Levenshtein edit distance between their string representations. We call the formal *implication distance* (ID) and the latter Levenshtein distance (LD). Theorem proving, using an SMT solver [19], determines whether one invariant is logically implied by another, while Levenshtein distance compares two sets of invariants syntactically.

Finally, we use hierarchical clustering to group programs based on a distance calculation. We use the Unweighted Pair Group Method with Arithmetic Mean (UPGMA) clustering algorithm (e.g., [47]). UPGMA takes as input a distance matrix and identifies clusters that minimize the average cluster diameter. Critically, since the UPGMA algorithm does not require the distance

matrix to be symmetric, we can use either LD or ID measurements to group programs. A cluster of identical patches can be inspected for a functional trust assessment by assessing any representative of it. On the other hand, the distances between clusters and the invariants on which they differ can communicate the effects of a patch to developers.

Program Repair V&V Reduction Results. We evaluated PatchPart on a set of 5 programs from the ManyBugs [45] and 7 programs from the Defects4J [37] benchmarks. We used 50 GenProg [45] patches for each C defect and 20 ARJA [85] patches for each Java defect (repair tools are often language-specific but our approach is agnostic).

For each program we studied, candidate patches were easily distinguished from the original program and relatively few invariants differentiated the repairs from the original, supporting our hypothesis that PatchPart can provide a concise assessment of the key semantic elements of a proposed repair, allowing a developer to quickly check that a repair retains required functionality.

Surprisingly, we found that Levenshtein Distance performs almost as well as Implication Distance for our use case, which reduces computational cost significantly. It is worth noting that we do not attribute a measure of importance to each invariant, and instead assume each invariant is equally meaningful. Regardless, because LD is significantly less expensive to compute, it is relevant in timed vehicular mission settings.

TRMO's PatchPart algorithm can successfully categorize patches based on their formal invariants. For each group of patches we can determine a hierarchy relationships, with each layer of the hierarchy containing more patches and representing a broader, more abstract set of features. On average, our evaluations found [14] this to lead to a reduction of patch classification effort by 50%: even when requiring distance-zero invariant sets of equivalent patches, the clusters were large enough to save developer inspection effort.

Ultimately, we found that changes in invariants characterize patch differences, that Levenshtein Distance reduces the cost of comparing invariant sets significantly, that clustering of semantically similar patches reduces validation costs on average by 50%, and that hierarchical clustering reveals how the clusters are related.

3.6.3 Proofs about Transformed Binaries

TRMO's hardening and diversity component, Zipr, can produce new binaries that are more difficult — or even impossible — to attack (Section 3.3). A key correctness condition for such tools is that, except when under attack, each new binary should have the same user-visible behavior as its corresponding original binary. We developed an initial capability for formally proving equivalence of such transformed binaries and applied it to prove the equivalence of certain exemplar binaries transformed by Zipr. This work was done on Intel x86 binaries, but a similar approach should work for other architectures given suitable machine models, as described below.

To reason about x86 binaries, we use the formal model of the x86 processor in the language of the ACL2 theorem prover [6] originally developed at the University of Texas at Austin and recently extended by Kestrel personnel [15, 30], as well as Kestrel's Axe tool [74, 75]. Our process for equivalence checking has three steps:

- First, we apply the Axe x86 Lifter to "lift" the original binary's functionality into a higher-level logical representation.
- Second, we lift the transformed binary similarly.

- Third, we apply a formal tool, such as the ACL2 prover or Axe equivalence checker, to prove equivalence of the original and transformed binaries.

This process increases confidence that the changes to the binaries made by TRMO transformations do not introduce errors that break the binaries' desired functionality. This is one approach for giving operators trust in the continued correct deployment of a system after resiliency operations (in this case, hardening transformations) have been applied.

Lifting Into Logic. To support reasoning about transformed binaries, we use the technique of *decompilation into logic* [2, 53, 74, 75], in which low-level code (binary or bytecode), is "lifted" into a higher-level logical representation. The lifted representation clearly indicates how executing the program affects the machine state (e.g., which memory writes are executed). This provides a basis for performing a rigorous equivalence proof of the behavior of the original and transformed binaries, showing that they are equivalent under normal (non-attack) conditions. The lifted representation may also support proving that the transformation fixes a given vulnerability (e.g., that unexpected malicious inputs no longer cause bad behaviors, or that undesirable information flows are no longer possible).

Formal x86 Machine Model. The core Axe tools are machine-independent, with the semantics of a particular processor or virtual machine being supplied by a formal model of that machine. The formal machine model provides the semantics of program being lifted. Each model is written in the language of the ACL2 theorem prover [6] and formalizes the state of the machine and the effect of each instruction on that state. Axe has so far been used with models of the JVM (Java Virtual Machine) [74] and the x86 processor [30]. The x86 model specifies the state of the x86 processor, including registers, flags, and a byte-addressable memory. For each of the hundreds of supported 32-bit and 64-bit instructions, a "semantic function" specifies precisely how executing the instruction affects the state, including operand fetching, the computation performed on the operands, how the result is written back, the various error checks performed, how much to advance the instruction pointer, and so on. Branches and subroutine calls and returns are modeled straightforwardly in terms of how they affect the instruction pointer and call stack. The model defines a "step" function, which fetches the next instruction and dispatches control to the appropriate semantic function, and a "run" function, which executes a series of instructions until the specified program or subroutine exits. The "run" function is the basis for Axe's symbolic execution, described next.

A formal machine model is often executable, allowing it to be validated by execution, comparing it against a hardware processor. In fact, the x86 model supports very efficient execution (up to 3MHz in one mode — note that this is a formal model running in a theorem prover). This allows the model to be validated by co-simulation against a real x86 processor.

The Axe Lifter. The Axe Lifter uses *symbolic execution* (e.g., [11, 39, 63]), to capture the meaning of a program in a higher-level logical form. Starting with a call of the model's run function that applies the target program to *arbitrary inputs*, rewrite rules are applied to repeatedly step (execute the next instruction) and simplify the resulting state. The result is a large symbolic term that represents the program's effect on the machine state, as a function of its inputs. For example, a program that adds two 32-bit numbers `x` and `y` and writes their sum to a given memory location,

might have, in the lifted representation, a memory write of the symbolic sum `(bvplus 32 x y)` at the given address.

The lifted representation abstracts away many details, such as instruction encodings, the successive values of the instruction pointer and, often, the use of temporaries (cf. [16]). The values of specific state components can be projected out of the final symbolic state if desired, or the entire symbolic state can be returned.

The process above works well for code segments that do not include loops or recursion. In particular, conditional branches are supported; the lifted result will then contain `if-then-else` operators, being careful to avoid path state space explosions when many conditional branches are present.

Axe uses two techniques to handle loops. First, loops may be "unrolled" when their iteration counts are fixed or can be bounded. This produces a large, loop-free term that captures the effect of the loop. When loops cannot be unrolled, Axe can lift an entire loop into a recursive logical function, a call of which is spliced into the ongoing symbolic execution. Contrast this with traditional symbolic execution, which would produce a separate path for each possible iteration count of the loop, leading to path explosion. The result of lifting contains a new logical function for each loop, including nested loops.

The unrolling technique has been used for a variety of programs, including the AES block cipher, whose main loop has 10 iterations. For the AESLightEngine Java class, Axe produces a term with 47,898,065,689,733,522 subexpressions, of which 3,646 are unique (Axe uses a compact term representation in which each unique subterm is represented only once). The term represents the ciphertext as a function of the plaintext and the key. Most details of the JVM's operation, including the operand stack, are simplified or projected away, and all subroutine calls are effectively inlined. The lifted representation includes calls to only 7 different functions, all of which operate on bit-vectors or arrays of bit-vectors (e.g., it includes 1,696 calls to BVXOR). Axe lifts this AES implementation in about 6 seconds.

The Axe technique of lifting loops into recursive functions has been applied to implementations of the SHA-1 and MD5 hash functions, since their loops cannot be unrolled, and to various other programs in the DARPA MUSE project [17].

Proofs About Transformed Binaries We demonstrated our equivalence proof process on several simple example x86 binaries. These evaluation programs include common features such as arithmetic, introducing subroutines, conditional branches, temporaries, and so on. They were designed to be indicative and show that our tools can handle a variety of program features. We considered ten programs and subjected them to transformations that moved around code our subroutines, introduced stack canaries, and the like. In all ten cases we were able to prove that the transformed programs were equivalent to the originals in terms of user-visible behavior.

Proofs After Changing the Stack Layout and Adding Canaries. To illustrate one concrete example, we consider TRMO's Zipr tool's stack layout transformation (SLX). SLX produces binaries that include special stack *canaries* — values on the stack which should not change and whose modification indicates the presence of a stack-smashing attack.

The transformed binary is significantly different from the original, but our formal approach is able to prove that those differences do not change normal execution. To provide some insight into

how this approach plays out, we present the result of lifting a simple function, `add1`, into a logical representation after applying the SLX transformation:

```
1 (defun-nx add1-transformed (x86)
2  (set-eip
3   (read-from-segment 4 (esp x86) 2 x86)
4   (set-eax
5    (bvplus 32
6          (read-from-segment 4 (+ 8 (esp x86)) 2 x86)
7          (read-from-segment 4 (+ 4 (esp x86)) 2 x86))
8    (set-edx
9     (read-from-segment 4 (+ 4 (esp x86))
10                  2 x86)
11    (set-esp
12     (+ 4 (esp x86))
13     (set-ebp
14      (bvchop 32 (ebp x86))
15      (write-to-segment        ; SLX transformation
16       4 (+ -232 (esp x86))    ; note write to ESP-232
17       2 (+ -4 (esp x86))
18       (write-to-segment
19        4 (+ -228 (esp x86))
20        2 172977
21        (write-to-segment
22         4 (+ -224 (esp x86))
23         2
24         (read-from-segment 4 (+ 4 (esp x86))
25                            2 x86)
26         (write-to-segment
27          4 (+ -220 (esp x86))
28          2 0
29          (write-to-segment
30           4 (+ -4 (esp x86))
31           2 (bvchop 32 (ebp x86))
32           (set-flag
33            :af 0
34            (set-flag
35             :cf 0
36             (set-flag
37              :of 0
38              (set-flag
39               :pf 1
40               (set-flag
41                :sf
42                0 (set-flag :zf 1 x86)))))))))))))))))
```

This lifted logical formula captures a key aspect of the transformation: the introduction of a write through the stack pointer (`ESP`) on lines 15–17. In addition, the transformation sets all the x86 status flags to constant values. If we clear the flags and zero out the parts of the stack written by both programs (i.e., the canary), we can prove equivalence:

```
1 (defthm equivalence
2  (implies
3   (x86p x86)
4   (equal (clear-all-flags
5         (write-to-segment 12 (+ -16 (esp x86)) *ss* 0
6          (write-to-segment 16 (+ -232 (esp x86)) *ss* 0
7           (add1 x86))))
8        (clear-all-flags
9         (write-to-segment 12 (+ -16 (esp x86)) *ss* 0
10         (write-to-segment 16 (+ -232 (esp x86)) *ss* 0
11          (add1-transformed x86)))))))
```

This formally captures a particular notion of correctness: unless your execution depends on the value of the canary, the binary procedures `add1` and `add1-transformed` are the same. Since

only stack-smashing attacks should influence the canary value, this gives formal confidence that normal mission operation should be unaffected by this transformation.

3.6.4 Formal Methods Summary

We used formal methods to provide additional evidence in support of the trusted operation of a TRMO system. We applied three different approaches to three key TRMO components.

First, we supported the Continuous Monitoring component during deployment (Section 3.4) via an architectural proof of our dual controller architecture. A critical correctness property of our system is that an attack should not be able to subvert our detection and repair mechanisms. In particular, while attacks may compromise the feature-rich and outward-facing locomotion controller, the trusted controller should control the autonomous vehicle's sensors and actuators during attack and recovery. We formally modeled our protocol and control handoffs, proving that exactly one controller controls the sensors and actuators at a given time.

Second, we supported the Code Repair component (Section 3.5) by reducing the verification and validation costs associated with automated program repair. Automated program repair can produce many candidate patches for the same defect — and not all patches are of equivalent quality. While a lower-quality patch that allows the vehicle to "fight through" the attack and complete the mission may be acceptable in the short term, ultimately operators may wish to inspect candidate patches and choose the best one to deploy or augment. We developed an approach based on distance metrics in invariant spaces to characterize the functional behavior of patches. This allowed us to cluster and partition them so that the number of patches that needed to be manually inspected (i.e., the amount of code that needed to be read carefully for V&V) was reduced by 50%, on average.

Third, we supported the Diversification and Hardening component (Section 3.3) via formal proofs of transformation correctness. Our diversity and hardening transformations should be semantics preserving in the sense that normal operations are the same with or without the transformation — they only differ in the presence of attacks (e.g., by detecting or shutting down the attack, rather than being compromised by it). We demonstrated that it is possible to prove pre- and post-transformation binary methods equivalent. We did so by lifting each method to a formal, logical representation capturing its low-level x86 meaning and operations, and proving implications and equivalences on those logical formulae.

4 Results and Discussion

We evaluated TRMO technologies in multiple ways. First, we assessed individual components, such as Diversification and Hardening, or Analysis and Modeling, in isolation to assess their overheads and efficacies. Since these components are "always on", in a sense (unlike Repair, which is only invoked when an attack is detected), their performance profiles are critical to overall mission deployment.

Second, we evaluated our system end-to-end, including the Repair component, on a series of attack scenarios. This included both software simulations and live demonstrations of autonomous vehicles, both x86 ground-based rovers and ARM quadcopters, in front of government personnel.

Attack scenarios and evaluations were gathered and overseen by a government-provided Red Team, Assured Information Security.

Key evaluation and discussion aspects include:

- An evaluation of TRMO's Diversification and Diversification component (i.e., of Helix and Zipr), in Section 4.1.
- An evaluation of TRMO's Analysis and Modeling component (i.e., of CMT), in Section 4.2.
- An end-to-end evaluation of TRMO's ability to detect, repair and fight through cyberattacks for autonomous vehicles (i.e., of our system as a whole, including Darjeeling), in Section 4.3.2.
- A discussion of the academic and industrial impact of the research developed under this project, in Section 4.3.4.

4.1 Evaluation — Diversification and Hardening

Because the Helix and Zipr technologies used by TRMO for diversification and hardening could also be used independently, we assess and evaluate their performance with respect to a number of metrics.

4.1.1 Evaluation — Diversification and Hardening — Preprocessing Time

Table 5: Helix toolchain: time to transform the ArduPilot X86 binary (1.3MB).

Helix Toolchain Step	Time to process Helix-as-a-Service (3GHz)	Time to process Local laptop (2.9GHz, SSD)
Initial IR analysis	2 mns	1+ mn
SLX	55 sec	30 sec
SCFI	40 sec	16 sec
GLX	30 sec	24 sec
Zipr (emit binary)	45 sec	17 sec
Total	~5 mns	2 mn 40 sec

During the period of performance, we streamlined the time required for Helix to analyze and transform binaries. Previously, each stage or transformation in the Helix pipeline would read the current state representation of the binary from the Helix database into memory, perform the transformations, and then write the state back out to the database. We observed that accessing the database for reading and writing represented a significant amount of time in relation to the overall processing time. To improve performance, we restructured the Helix toolchain to coalesce

database accesses whenever possible. Thus, only the first and last transformations need to access the database (the first transformation reads from the database, the last transformation writes the final state to the database, while intervening transformations use in-memory data structures).

Table 5 shows final results from this restructuring for processing `ardupilot`, the binary (1.3MB) used to control the X86-based ground rover used in the live demonstrations. Under the configuration of "Helix-as-a-service", representative of a cloud-based scenario, transforming the binary took approximately five minutes. Under the configuration of "Local laptop with SSD", representative of a high-powered autonomous vehicle with a solid state drive, our toolchain required only two minutes and forty seconds. Reducing database access time and using solid-state storage (instead of spinning hard drives) has a dramatic impact on processing time.

Processing the equivalent TRMO ARM quadcopter `ardupilot` binary with only the initial IR analysis and Zipr steps took 12 minutes on a Raspberry Pi 3+. Salient features of the Raspberry include its limited main memory (1GB), slow CPU (1.4GHz), and I/O channels limited by a USB 2.0 connection (300 MBits/s). This represents a worst-case scenario for our preprocessing approach.

We note that the Helix toolchain supports cross-analysis and production of binaries. In operational scenarios where fast transformations are desired, Helix can be deployed as a service in a cloud environment with fast X86 CPUs and solid-state drives to transform *both X86 and ARM binaries*. Furthermore, as each binary can be generated independently, it would be natural to scale horizontally across a cloud environment to support the generation of tens-of-thousands of Helix protected binaries to support multiple missions.

4.1.2 Evaluation — Diversification and Hardening — Runtime Overheard

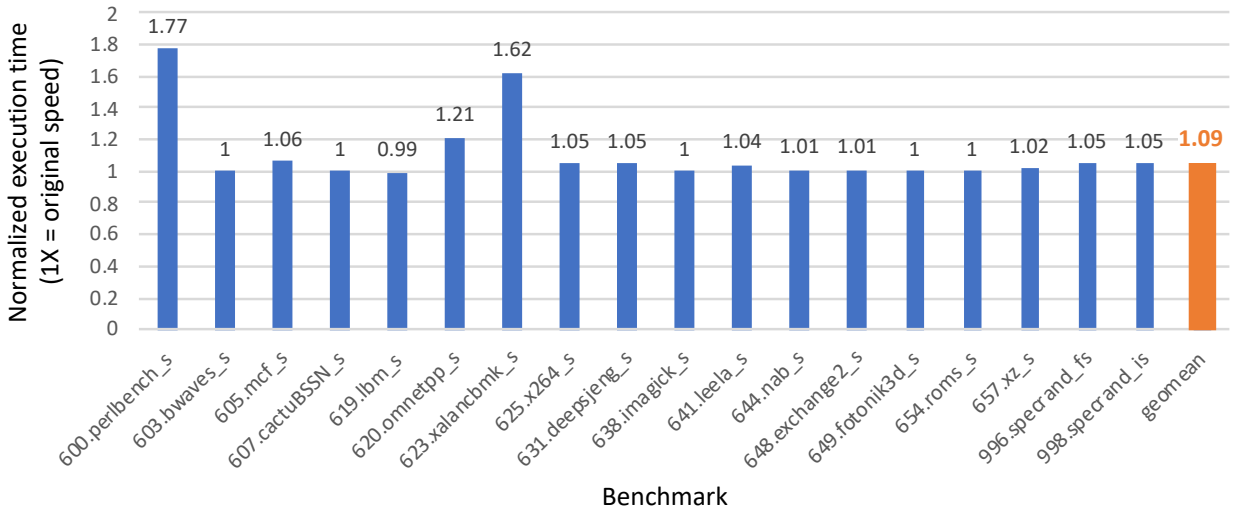

Figure 11: Runtime performance overhead of Zipr/BILR on ARMv8 SPEC2017 benchmarks.

In Figure 11, we show performance overhead for Zipr using the BILR transformation on the SPEC 2017 benchmark suite of indicative programs for 64-bit ARM code. On average, Zipr/BILR incurs 9% overhead. While most benchmarks showed modest overheads (15 or 18 benchmarks incurred

less than 6% overhead), two benchmarks, `perlbench` and `xalancbmk`, clearly stand out with 77% and 62% overhead respectively.

In addition to Figure 11, which shows ARM performance, we also investigate x86 performance in Figure 14. There, Zipr/BILR performance for X86 on SPEC 2006 is shown with blue bars, and both `perlbench` (35% overhead) and `xalancbmk` (50% overhead) also stand out in terms of overhead. While these performance numbers are not directly comparable (ARM vs. x86), we note that both benchmarks make heavy use of indirect branches, and are known to be challenging for binary rewriters.

The X86 version of Helix has been heavily optimized throughout multiple Air Force and Department of Defense Projects, including the original Air Force MURI that was the genesis for the Helix toolchain, the DARPA Cyber Grand Challenge, and the DARPA Cyber Fault-tolerance Attack Recovery program. With further work in optimizing the ARM platform, we expect to achieve similar performance gains as with the X86 platform.

4.1.3 Evaluation — Diversification and Hardening — Supporting Exceptions

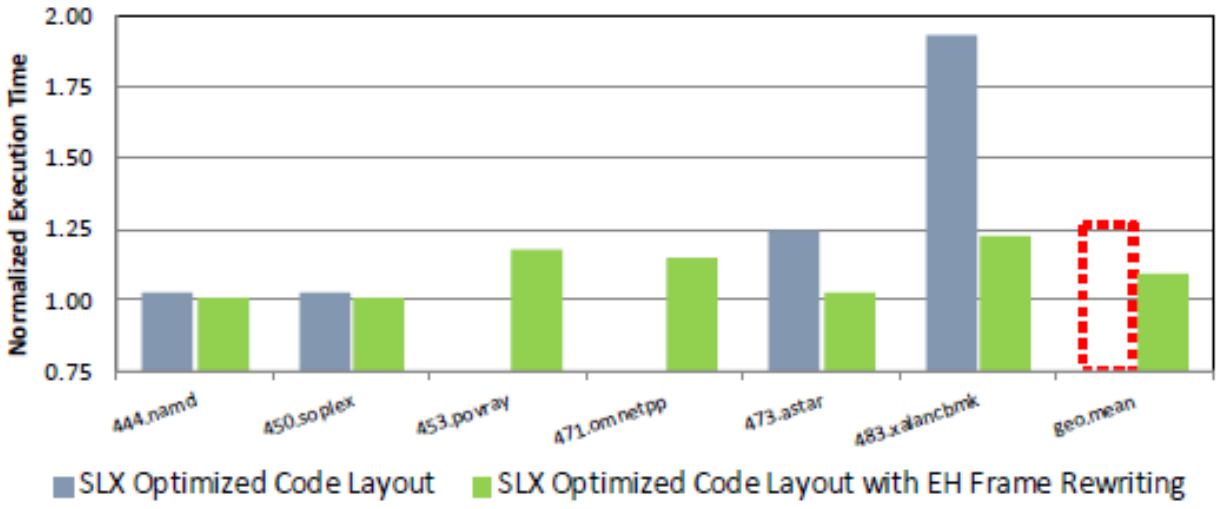

Figure 12: Exception handling information recovery enables safer and faster SLX transformations. The dashed red box indicates what the geometric mean overhead of the default SLX approach would be if it applied to the povray and omnetpp.

Work on static binary rewriting has traditionally ignored exception handling, a critical feature in languages such as C++. As part of this effort, we showed that recovering and exploiting exception handling information actually results in both *safer* and *faster* transformations [35].

Figure 12 shows the performance overhead of SLX, our stack transformation, without exception handling (blue bars) and with exception handling (green bars) for several C++ programs in the SPEC CPU2006 benchmark suite. Note that neither `povray` nor `omnetpp` have grey bars. Both benchmarks dynamically throw and catch exceptions as part of their normal operation, and thus, stack-based transformations *must* take into account and update exception handling information as

part of the rewriting process. In terms of performance, incorporating and supporting exception handling information reduces the overhead of SLX from 26% down to 10%.

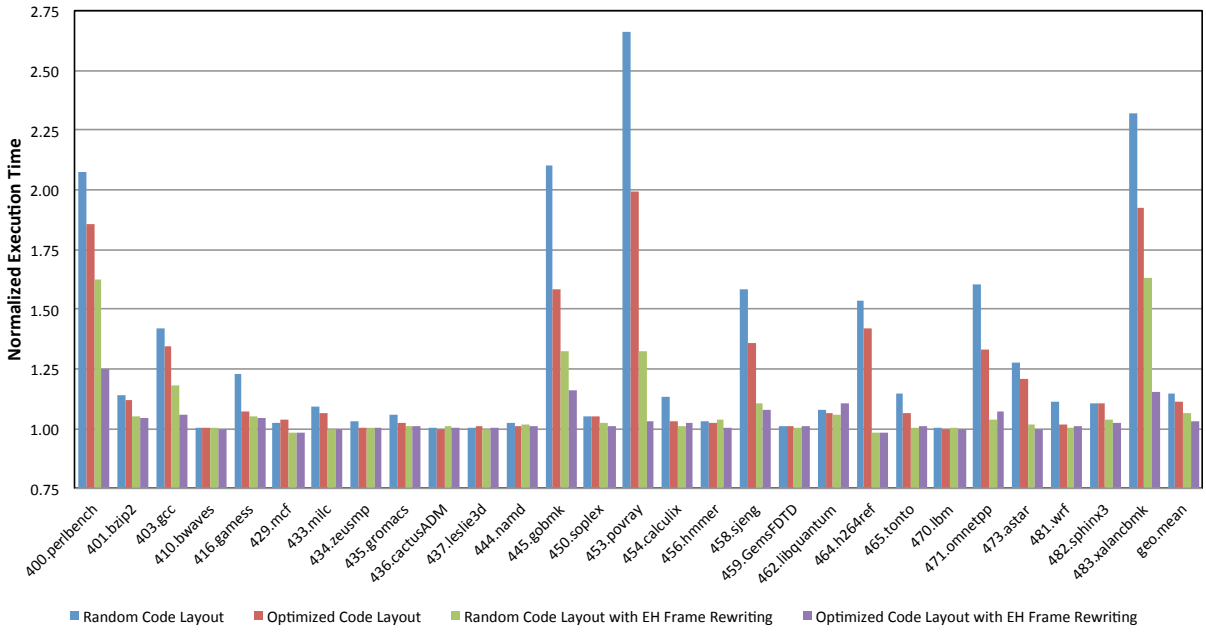

Figure 13: Performance overhead of Zipr with exception handling support of SPEC CPU2006 benchmarks.

To isolate and highlight performance improvements enabled by incorporating exception handling information, Figure 13 compares various Zipr/BILR configurations with no additional transformations. The green and purple bars highlight Zipr performance with exception handling support for both a random Zipr code layout policy (favoring diversity) and an optimized code layout policy (favoring performance), resulting in overall average overheads of 7% and 3% respectively.

These numbers echo a general Helix theme that more precise reverse engineering of binaries results in safer, more secure, and faster rewritten binaries. Incorporating exception handling in Helix resulted in the breakthrough capability of only incurring 3% as the baseline cost for rewriting binaries.

4.1.4 Evaluation — Diversification and Hardening — Third-Party ROP

Table 6: Third-party case study: Zipr high-entropy diversity engine *significantly* outperforms other state-of-the-art tools in reducing ROP gadgets. Table reprinted from [7].

				Reduction (%) of Turing-complete (TC) gadgets in 7 TC categories (MIN-FP | EX-FP)							
Tools	Granularity	⇓ (%) MIN-FP	⇓ (%) EX-FP	Memory	Assignment	Arithmetic	Logical	Control Flow	Function Call	System Call	TC Preserved?
Applications											
Zipr	Inst.	**80.91**	**88.45**	**100** | 93.5	61.9 | 91.5	**100** | 86.3	57.5 | 82.1	66.0 | 88.7	73.1 | 92.5	83.33 | 0	✗*
SR	FB	40.28	36.53	3.6 | 21.0	10.8 | 42.9	14.7 | 9.8	35.5 | 36.2	23.4 | 29.3	25.0 | 48.4	0 | 0	✓
MCR	FB & Reg.	37.19	34.81	-16.7 | 25.6	-4.4 | 23.0	22.0 | 38.8	2.4 | 28.8	40.5 | 59.2	14.0 | 63.7	80.0 | 0	✓
CCR	BB	27.02	38.77	2.3 | 31.7	4.4 | 41.2	12.2 | 24.4	4.8 | 26.4	56.0 | 71.2	30.9 | 61.4	0.0 | 0	✓
Libraries											
Zipr	Inst.	**91.63**	**85.42**	**94.4** | 91.4	67.2 | 89.1	**96.8** | 88.1	83.5 | 89.0	65.4 | 89.1	62.5 | 86.7	66.67 | 0	✗*
SR	FB	23.54	37.91	23.5 | 29.3	19.2 | 40.4	31.5 | 43.1	48.9 | 43.2	47.7 | 56.1	36.6 | 39.9	22.91 | 0	✓
MCR	FB & Reg.	6.34	37.77	24.1 | 37.5	30.2 | 39.6	56.3 | 55.9	45.7 | 45.4	37.0 | 54.1	43.4 | 42.3	66.67 | 0	✓
CCR	BB	10.89	33.66	9.7 | 26.5	11.1 | 46.4	22.6 | 35.9	21.9 | 39.8	25.9 | 45.6	23.2 | 44.6	50.0 | 0	✓

* For Zipr, TC is not preserved for minimum footprint gadgets, but TC is preserved for extended footprint gadgets.

A major defensive capability provided by Helix is the high-entropy diversification of binaries using Zipr/BILR. The rationale for diversification is to make the attack surface of software unpredictable and therefore raise the difficulty level for carrying out successful attacks.

During the period of performance, researchers at Virginia Tech accessed and evaluated the Helix toolchain. After we provided the code (with Helix on a Docker image), we had no further interactions with them. Several months later, the Virginia Tech group sent us a link to their paper, *Measuring Attack Surface Reduction in the Presence of Code (Re-)Randomization*, comparing various diversity tools and techniques for defeating return-oriented programming (ROP) attacks [7].

Table 6 is a table extracted from their paper. Of all the tools compared, Zipr is the only one that can operate directly on stripped binaries. Other tools required symbol information, or access to the source code and/or compiler toolchain, making them ill-suited for deployment scenarios where only stripped binaries are available.

Practical deployment issues aside, the table highlights the percentage reduction in gadgets available to attackers for carrying out a ROP attack. Zipr achieves significant attack surface reduction (the Zipr numbers are boldfaced in the authors' paper) for application and library binary code, both in absolute terms and compared to other tools. The last column in the table indicates whether Turing Completeness (TC) is maintained from the perspective of the attacker. If TC is maintained, then the set of available ROP gadgets post-diversification are theoretically sufficient and powerful enough to carry out arbitrary attacks. Again, Zipr is explicitly highlighted as it was the only tool able to break TC under one of the two gadget category evaluated in the case study.

4.1.5 Evaluation — Diversification and Hardening — Control-Flow Integrity

Control-flow Integrity (CFI) is a powerful technique for preventing arc-injection attacks. At a high-level, CFI dynamically enforces the control-flow graph embodied in the source code of a program.

Figure 14: Performance overhead of Zipr, Zipr+Original CFI, Zipr+New CFI, Zipr+New CFI Coloring for X86-64 on SPEC CPU2006.

If an attack results in a new edge in the control flow graph then CFI will halt program execution thereby thwarting the attack.

CFI is most often implemented as a compiler option and thus requires source code. Helix implements CFI for binaries and was deployed successfully during the Cyber Grand Challenge (CGC) [60] where it achieved the best defensive score. To protect the X86-based ground rover in TRMO, we made several enhancements:

- We extended our CFI implementation to X86-64 as CGC was based on a X86-32 bit platform.
- We optimized performance by making heavy use of executable nonces for enforcing CFI. Executable nonces enabled us to use CFI instrumentation that did not interfere with the hardware-optimized call/return stack.
- We improved our switch table detection heuristics. When a switch table can be analyzed to such an extent that indirect branch targets can only provably branch to well-known targets, then CFI instrumentation is not needed. Knowing when *not* to add instrumentation or security checks improves performance without sacrificing safety.
- We implemented CFI coloring, a technique which uses different values (colors) for identifying different indirect target branch sets. Coloring provides more security as indirect branches are further restricted based on their potential targets.

Figure 14 shows the performance overhead of various CFI configurations. The blue bar denotes baseline Zipr overhead (6% overhead). The dark orange bar denotes overhead for the original CFI implementation (15% overhead). The grey bar denotes our improved CFI without any coloring (11% overhead). Finally, the last orange bar denotes the performance of CFI with coloring (16% overhead).

In terms of security and performance tradeoffs, our new CFI implementation achieves 4% improvement in performance overhead for the same level of security as our original CFI (15% $\rightarrow$ 11%). For the multi-colored version, our new CFI achieves higher security at the cost of only an additional 1% (15% $\rightarrow$ 16%).

To the best of our knowledge, Helix is the only toolchain with efficient multi-colored CFI capabilities for stripped binaries.

4.1.6 Evaluation — Diversification and Hardening — Binary Fuzzing

Table 7: ZAFL outperforms other state-of-the-art binary fuzzers.

ZAFL	AFL-Dyninst	AFL-QEMU
ZAFL performance	+60%	+228%
ZAFL test cases	+48-78%	+150-835%
ZAFL unique crashes	+26-96%	+42-151%

TRMO incorporates proactive hardening and diversity combined with reactive automated program repairs when an attack is detected. Fuzzing of autonomous vehicle binaries to uncover latent vulnerabilities opens up the possibility of *proactive repair* of code, prior to mission deployment, in a two step process: (1) find crashing inputs, and then (2), feed crashing inputs to a repair process.

In a separate Air Force SBIR project, we developed ZAFL, a binary fuzzer that leverages Helix to insert fuzzing instrumentation for the American Fuzzy Lop (AFL) fuzzing platform. The efficiency of Zipr combined with an easy-to-use SDK for writing transformations resulted in ZAFL being the fastest binary fuzzer for AFL, as independently evaluated by another university [33].

All other things being equal, the effectiveness of fuzzing depends on raw performance (i.e., the rate at which random inputs can be generated and assessed). ZAFL leverages Helix's ability to inline instrumentation and layout efficient binaries.

Table 7 summarizes results for ZAFL compared to two other state-of-the-art AFL-based binary fuzzers, AFL-Dyninst and AFL-QEMU, on a set of real-world benchmarks comprising audio, video, and XML parsing applications. In fuzzing sessions over a 24 hour time period, ZAFL outperformed AFL-Dyninst by 60% and AFL-QEMU by 228% in terms of raw performance. ZAFL is able to generate 48–78% more unique test cases than AFL-Dyninst, and 150–835% more than AFL-QEMU. ZAFL also finds 26–96% more unique crashes than AFL-Dyninst, and 42–151% more than AFL-QEMU.

As a testament to the robustness of Helix and ZAFL, a fuzzing startup reached out to us and gave us a 143MB challenge binary to fuzz. The binary, `catboost`, is a well-known machine learning package developed by Yandex, Russia's equivalent of Google. We were able to download and fuzz `catboost`, and then discover and confirm an unknown heap overflow vulnerability, all before the startup was able to even build the binary with the standard AFL set of tools. This episode depicts clearly the advantage of working at the binary level. When dealing with source code for real-world software, hooking into the build system is non-trivial and its difficulty often under-appreciated. By working with binaries directly, Helix and its tools sidestep the difficulties inherent in dealing with source code.

4.2 Evaluation — Analysis and Modeling — CPU and Memory Overhead

Because the CMT technology used by TRMO for analysis and modeling could also be used independently for trust assessment and anomaly detection, we assess and evaluate its performance overhead separately.

CMT's monitoring and verification incur a runtime overhead. The overhead associated with monitoring and verifying locomotion comes primarily from the CPU resources required. By contrast, network overhead is negligible. On the quadcopter live demonstration system, a Raspberry Pi 3B+, CMT's monitoring and verification process utilized on average 5% of a single processor core. By contrast, the flight stack ArduPilot process utilized on average 17% of a single process core. Memory utilization on average was less than 5% of the total system memory. The locomotion component of CMT was well within overhead constraints to allow the flight stack room to use more resources as necessary, and to prevent resource congestion or alter mission operation.

CMT also made use of the Extended Berkeley Packet Filter trace API (Section 3.4.6) to inspect kernel traces. The chosen kernel traces are directly related to the model used to monitor and verify system operations that may be vulnerable to attacks. Here, overhead was measured against an Apache 2 server executing client HTTP requests, where the requests triggered execution, file

system, and memory operations on the server system. The baseline utilization of Apache 2 without monitoring used an average of 12.75% of a single processor core. Minimal critical traces, using Auditd to generate the traces for kernel function calls, required an average of 50.57% of a single processor core. This equates to an equivalent 12.64% of a four processor core Raspberry Pi 3B+ system. Memory utilization by the Auditd traces was also minimal, totaling less than 3% of the available 926 MB of main memory.

4.3 Evaluation — End-to-End

The ability to deploy Helix-transformed binaries, detect attacks with CMT, construct repairs on the fly, and ultimately enable fight-through capabilities for autonomous vehicles was demonstrated live in front of government personnel at the University of Michigan M-AIR drone testing facilities. In 2018, live demonstrations focused on ground-based x86-64 autonomous rovers. In 2019, live demonstrations focused on aerial ARM-32 quadcopters. In both cases, attack scenarios (e.g., security exploits, software defects, etc.) were furnished by a government-provided Red Team.

Following the rules of engagement and system assumptions (Section 3.2), attacks took the form of specially-crafted packets sent by a remote attacker. No cryptographic defenses were used (the attacker was assumed to have bypassed them). While the vehicle was completing a simple multiple-waypoint mission, attack packets were sent near particular locations, simulating an attacker with a directional antenna.

Except when attacks occurred, no discernible behavioral differences were observed during practice run and live demonstrations: the rover and quadcopter performed their missions without observable differences in terms of safety, speed, height, or battery drainage, even though TRMO components (such as Helix and Zipr provided hardening and diversity, or CMT providing analysis and monitoring) were present.

4.3.1 Evaluation — End-to-End — x86 Rover Demonstration

Table 8 shows the results of the Red Team evaluation of TRMO defenses applied to a live x86 rover autonomous vehicle defending against attack scenarios without recourse to external servers. The scenario identifiers (e.g., 1, 7, etc.) are as labeled during the Red Team engagement.[10] Beyond buffer overflows, which were important enough to merit multiple variations, the scenarios were unique.

[10]Other scenarios were made available for software simulation testing and assessment in preparation for the live evaluation.

Table 8: Red Team evaluation of TRMO defending a live x86 autonomous ground vehicle against attack scenarios with no external servers.

	Scenario Description	Hardening	Analysis	Repair	Time (m)
1	use-after-free		Detect	Repair	11.3
7	stack-based buffer overflow	Detect	Detect	Repair	10.0
10	infinite loop		Detect	Repair	9.8
12	segmentation fault	*Prevent*	–	–	–
15	integer overflow		Detect	Repair	8.5
k	arc-injection function pointer	Detect	Detect	Repair	17.0
g	stack-based buffer overflow	Detect	Detect	Repair	16.0
de	input sanitization and function pointers		Detect	Repair	11.0
do	double free		Detect	Repair	10.0
heldout1	integer cast		Detect	Repair	13.0
heldout2	overflow to exit		Detect	Repair	9.0

A "Detect" indicates that a TRMO defensive technology detected the attack and alerted the Repair component. For the Hardening column, this corresponds to a Helix/Zipr transformation (Section 3.3) hardening the software such that the vulnerability yields a detectable event when triggered instead of compromising the system. For the Analysis column, this corresponds to CMT (Section 3.4) detecting a violation of the learned trust model.

The "Prevent" for scenario 12 indicates that Helix/Zipr transformations were powerful enough to entirely mask the attack. The memory transformations employed rendered the attack packet harmless. As a result, there were no behavioral changes in the operation of the vehicle at all (and thus no trust policy violation for Analysis to detect and no invocation of the Repair component).

A "Repair" indicates that TRMO's Darjeeling technique for automated program repair (Section 3.5) was able to produce a patch. The Time column gives the time, in minutes, required to produce the patch using only the rover embedded vehicle hardware. This represents a worst-case evaluation in terms of repair latency. The average time required was 11.5 minutes.

We draw particular attention to the two "heldout" scenarios. While the TRMO team was given information about the other scenarios for training purposes as part of the Red Team engagement, the two held out scenarios were only made available after the TRMO defenses had been "locked in". In essence, they were evaluated live for the first time on the day of the demonstration. They thus illuminate TRMO's ability to successfully defend against entirely unknown or future-looking threats.

Other evaluation metrics were collected during this demonstration. The space overhead required to store additional files (e.g., TRMO defensive techniques, libraries, etc.) was 14–66% (or 1–4 GB). The executable overhead (i.e., for storing transformed and hardened binaries) was 4.6 MB (or $4.5\times$). Compared to the gigabytes required for the autonomous vehicle normally, the additional megabytes for the hardened binary were minute. The network overhead was 0 (repairs were conducted using the vehicle hardware). However, if remote hardware had been used, the network overhead would have been 1.31 MB. The network overhead is low because only the proof-of-concept attack packet and the patch (a small change to the executable) need be transmitted. Finally, the times reported above are the times required to construct the first repair. As noted

in Section 3.6.2, it is often desirable to produce multiple repairs (e.g., for diversity, to present to developers, etc.). The time to produce each additional repair beyond the first was $45s$, on average.

4.3.2 Evaluation — End-to-End — ARM Quadcopter Scenarios

Table 9: Red Team evaluation of TRMO defending an ARM autonomous quadcopter against attack scenarios with no external servers.

	Scenario Description	Hardening	Analysis	Repair
1	use-after-free	Detect	Detect	Repair
3	format string — information leak	Detect		
4	format string — crash	Detect		
5.1	stack-based buffer overflow	Detect	Detect	Repair
5.2	heap-based buffer overflow	Detect	Detect	Repair
9	ARM code injection	Detect	Detect	
10	infinite loop		Detect	Repair
12	segmentation fault		Detect	Repair
13	mathematical logic bug		Detect	Repair
14	denial of service		Detect	Repair
15	integer overflow		Detect	Repair
16	floating point exception		Detect	Repair
heldout17	concurrency		Detect	
heldout18	deadlock		Detect	
heldout19	delimiter injection		Detect	Repair
heldout20	double free	Detect	Detect	Repair
heldout21	function pointer overwrite	Detect	Detect	Repair
heldout22	memory leak			
heldout23	off-by-one	Detect	Detect	Repair
heldout24	SQL validation		Detect	
heldout25	uninitialized memory			

Table 9 shows the results of the Red Team evaluation of TRMO defenses applied to an ARM autonomous quadcopter. The scenarios are similar to those used for the x86 rover demonstration when possible, but adapted to the ARM architecture (e.g., x86 code injection attacks are replaced with ARM code injection attacks). Certain x86 vehicle exploits (e.g., "heldout1", "k", etc.) could not be adapted to ARM and were not evaluated. The nine final scenarios (labeled 17–25) "held out" were only made available after TRMO defenses had been "locked in", following the Red Team rules of engagement.

A "Detect" indicates that a TRMO defensive technology, such as Helix/Zipr (Hardening, Section 3.3) or CMT (Analysis, Section 3.4) was able to detect the attack. Note that TRMO as a whole detects the attack if either defensive layer detects the attack. TRMO thus detects 19 of the 21 (90%) of the attacks considered.

A "Repair" indicates that TRMO was able to entirely repair the defect or vulnerability (Section 3.5) and complete the mission. The end-to-end TRMO system thus provided "fight through" resiliency capabilities for 13 out of 21 (62%) of the Red Team quadcopter attack scenarios.

4.3.3 Evaluation — End-to-End — ARM Quadcopter Demonstrations

Four scenarios were selected for live flight demonstration in front of government personnel and Red Team members. Video recordings were made of live flights showcasing TRMO technologies: edited, narrated presentations of them are available.

Information Leak Attack. In this scenario, a remote attacker exploits a weakness in an on-board image server on the quadcopter, bypassing its session authorization routine to download sensitive images. This type of weakness is commonly referred to as "improper authorization" (Common Weakness Enumeration #285). By sending a carefully-crafted message, an attacker can bypass the image server's session authorization mechanism and exfiltrate sensitive data from the UAV.

TRMO successfully detected the attack before any files were exfiltrated. Helix/Zipr diversified the file server software used against memory-based attacks, and CMT's `inotify` handlers detected an unexpected file system access pattern. TRMO lands safely to construct a repair. The repair is constructed by injecting an existing permission check from elsewhere in the source into the section of faulty code; this prevents future exploits of the vulnerability. The mission is resumed and completes successfully. The Red Team measured that constructing the repair took 1.60 minutes.

Figure 15: Example frames from narrated videos, suitable for broad government audiences, showcasing TRMO defenses applied live to attacks. Left: information leak attack. Right: segmentation fault attack.

An edited video introducing and displaying this demonstration is available at https://www.youtube.com/watch?v=nYWF-OBd51w. See Figure 15 for an example.

Segmentation Fault Attack. In this scenario, a remote attacker exploits a weakness in the autopilot's handling of command and control messages. This type of weakness is commonly-referred to as "improper input validation" (Common Weakness Enumeration #20). By sending a carefully-crafted C2 message, an attacker can cause a control-flow jump (i.e., an arc-injection) to modify the autopilot's tasking to trigger a segmentation fault.

TRMO successfully detected the attack: Zipr's diversification prevents the arc-injection attack from succeeding by relocating the target code. TRMO lands safely and constructs a repair by disabling the errant run-time parameter handling logic and closing the attack vector. The mission then resumes and completes successfully. The Red Team measured that constructing the repair required 2.20 minutes.

An edited video introducing and displaying this demonstration is available at https://www.youtube.com/watch?v=ZBSexhSHHcY.

Sensor Attack. In this scenario, a remote attacker exploits a weakness in the image analysis routine's handling of sensed imagery data. This weakness is commonly referred to as "improper handling of unexpected data" (Common Weakness Enumeration #241). By exposing the image sensor to concentrated red hues, the attacker can force the Robot Operating System- and ArduPilot-based autonomous aerial vehicle controller architecture into inducing a fail-fast landing.

TRMO successfully detected the attack, landed safely to construct a repair, and then resumed and completed the mission. The repair replaces the malicious function call with a safe function call, repurposed from elsewhere in the program. The Red Team measured that constructing the repair took 2.05 minutes.

An edited video introducing and displaying this demonstration is available at https://www.youtube.com/watch?v=L3LP-0omccc.

Logic Bug. In this scenario, a remote attacker exploits an implementation of a weakness in the autopilot's handling of runtime device parameter settings. This type of weakness is commonly referred to as "assigning instead of comparing" (Common Weakness Enumeration #481). By sending a carefully-crafted `set-parameter` message, an attacker causes a crash failure in the quadcopter's autopilot software (by triggering a floating-point exception which causes a segmentation fault).

TRMO successfully detected the attack, landed safely to construct a repair, and then resumed and completed the mission. The repair effectively disables the errant run-time parameter handling logic and closes the attack vector. The Red Team measured that constructing the repair took 2.68 minutes.

An edited video introducing and displaying this demonstration is available at https://www.youtube.com/watch?v=kRpQStJ4UdI.

4.3.4 Discussion — External Impact

Over the course of multiple years, the TRMO effort led to significant advances in the state of the art. For example, automated program repair advances scaled the technique from constructing repairs in 90 minutes to on the order of 5 minutes; static binary rewriting and hardening were adapted to handle ARM and C++; and trust assessment was extended to handle autonomous vehicle semantics.

Academically, techniques associated with the project have led to multiple publications. For brevity, we highlight only the most influential, including six that received Distinguished Paper Awards:

1. A Genetic Programming Approach to Automated Software Repair [24]

2. Automatically Finding Patches Using Genetic Programming [83]

3. GenProg: A Generic Method for Automated Software Repair [46]

4. Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair [72]

5. Using Dynamic Analysis to Discover Polynomial and Array Invariants [58]

6. Using Execution Paths to Evolve Software Patches [59]

In addition, two of those papers also received Ten-Year Most Influential Paper Awards [24, 83], highlighting their "test of time" importance and impact on the academic community.

The Helix/Zipr binary hardening, diversifying and rewriting techniques associated with TRMO took second place overall in the DARPA Cyber Grand Challenge (CGC), and obtained the highest defensive score overall [32, 60].

5 Conclusions

This report describes the *Trusted Resilient Mission Operation* (TRMO) system architecture, which focuses on efficient, trusted, resilient techniques that allow autonomous vehicles to "fight through" an attack and continue a mission. TRMO integrates a set of static and dynamic protective technologies for Intel- and ARM-based rover and quadcopter platforms. This report describes those techniques and their evaluations.

TRMO's Diversification and Hardening components, Helix and Zipr, provide composable, low-overhead transformation (Section 3.3) that detect and prevent many classes of attacks (Table 1), and support critical languages like C++ (Section 3.3.2). These defenses incur only a 9% runtime overhead on ARM (Figure 11) but detected 43% of Red Team-furnished attacks against an autonomous quadcopter (Table 9).

TRMO's Analysis and Modeling component, CMT, uses a combination of readily-available telemetry information (Table 3) and operating system support (Section 3.4.6) to detect trust policy violations (anomalies and attacks). This defense incurred only a 5% runtime overhead on ARM (Section 4.2), and after training detected 81% of Red Team-furnished attacks against an autonomous quadcopter (Table 9).

TRMO also included efforts to support operator trust in the deployment of a resilient system. We formally proved properties relating to the isolation between TRMO defenses (e.g., attack detection, repair, etc.) and vulnerable payloads (Section 3.6.1). We introduced invariant-based approaches to reduce the manual verification and validation costs associated with inspecting multiple patches by 50% (Section 3.6.2). We formally proved that certain hardening and diversification transformations result in programs that are equivalent on benign inputs but differ when attacked (Section 3.6.3).

Finally, TRMO's Repair component, Darjeeling, used search-based techniques, in the style of GenProg, to automatically construct patches for detected defects (Section 3.5). A combination of pre-mission analyses (Section 3.5.3) and in-mission optimizations (Section 3.5.4) admit the efficient generation of repairs. TRMO was able to produce repairs for 62% of Red Team-furnished

attacks against an autonomous quadcopter (Table 9). Red Team measurements of repair construction were under three minutes (Section 4.3.3).

The TRMO effort developed, integrated and evaluated techniques to provide trusted and resilient operation for autonomous vehicle missions via a combination of static defenses, models and proofs as well as dynamic monitoring and repair. In the final Red Team evaluation using an ARM quadcopter, TRMO detected 90% of attacks and constructed a repair for 62%, allowing the vehicle to fight through and complete those mission successfully.

References

[1] American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[2] The Axe toolkit. https://github.com/acl2/acl2/tree/master/books/kestrel/axe.

[3] Capstone, the ultimate disassembler. http://capstone-engine.org.

[4] Qemu, the fast! processor emulator. http://qemu.org/.

[5] DARPA Cyber Grand Challenge. http://www.cybergrandchallenge.com, 2016.

[6] The ACL2 theorem prover. http://www.cs.utexas.edu/~moore/acl2.

[7] M. S. Ahmed, Y. Xiao, G. Tan, K. Snow, F. Monrose, and D. D. Yao. Measuring attack surface reduction in the presence of code (re-)randomization. https://arxiv.org/pdf/1910.03034.pdf, 2019.

[8] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *International Symposium on Software Testing and Analysis*, pages 177–188, 2016.

[9] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The Plastic Surgery Hypothesis. In *International Symposium on Foundations of Software Engineering*, FSE '14, pages 306–317, 2014.

[10] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering (TSE)*, 41(5):507–525, 2015.

[11] J. Berdine, C. Calcagno, and P. W. O'hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems*, APLAS '05, pages 52–68, 2005.

[12] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazires, and D. Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, pages 227–242, May 2014.

[13] A. A. Capiola, A. D. Nelson, C. Walter, T. J. Ryan, S. A. Jessup, G. M. Alarcon, R. F. Gamble, and M. D. Pfahler. Trust in software: Attributes of computer code and the human factors that influence utilization metrics. In *HCI International 2019 - Posters - 21st International Conference, HCII 2019, Orlando, FL, USA, July 26-31, 2019, Proceedings, Part I*, pages 190–196, 2019.

[14] P. Cashin, C. Martinez, W. Weimer, and S. Forrest. Understanding automatically-generated patches through symbolic invariant differences. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 411–414, 2019.

[15] A. Coglio and S. Goel. Adding 32-bit mode to the acl2 model of the x86 isa. *Electronic Proceedings in Theoretical Computer Science*, 280:77–94, 10 2018.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[17] DARPA. Mining and Understanding Software Enclaves (MUSE), 2014. https://www.darpa.mil/program/mining-and-understanding-software-enclaves.

[18] J. W. Davidson, C. L. Coleman, J. D. Hiser, and A. Nguyen-Tuong. System, method and computer readable medium for space-efficient binary rewriting, 2016. US Patent US20170371635A1.

[19] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[20] M. Elnaggar, J. D. Hiser, T. X. Lin, A. Nguyen-Tuong, M. Co, J. W. Davidson, and N. Bezzo. Online control adaptation for safe and secure autonomous vehicle operations. In *Adaptive Hardware and Systems (AHS), 2017 NASA/ESA Conference on*, pages 101–108. IEEE, 2017.

[21] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007.

[22] K. Fisher, J. Launchbury, and R. Richards. The HACMS program: using formal methods to eliminate exploitable bugs. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 375(2104), 2017.

[23] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[24] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference*, pages 947–954, 2009.

[25] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[26] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

[27] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. In *International Conference on Software Maintenance*, pages 1–10, 2010.

[28] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, Tucson, AZ, USA, June 9–11, 2008.

[29] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*. Internet Society, 2008.

[30] S. Goel, W. A. H. Jr., M. Kaufmann, and S. Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *Proc. 14th Conference on Formal Methods in Computer-Aided Design (FMCAD 14)*, 2014.

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

[32] W. Hawkins, J. D. Hiser, A. Nguyen-Tuong, M. Co, and J. W. Davidson. Securing binary code. *IEEE Security & Privacy*, 15(6):77–81, 2017.

[33] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson. Zipr: Efficient static binary rewriting for security. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 559–566, 2017.

[34] T. Helmuth, L. Spector, and J. Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, 2015.

[35] J. Hiser, A. Nguyen-Tuong, W. Hawkins, M. McGill, M. Co, and J. Davidson. Zipr++: Exceptional binary rewriting. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 9–15. ACM, 2017.

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

[37] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[38] M. Kaufmann and J. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26:161–203, 2001.

[39] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[40] D. Kim, J. Nam, J. Song, and S. Kim. Automatic Patch Generation Learned from Human-written Patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[41] X.-B. D. Le, D. Lo, and C. Le Goues. History Driven Program Repair. In *International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, 15 Mar. 2016.

[42] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, March 2018.

[43] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, 2012.

[44] C. Le Goues, S. Forrest, and W. Weimer. Representations and operators for improving evolutionary software repair. In *Genetic and Evoluationary Computation Conference*, GECCO '12, pages 959–966, 2012.

[45] C. Le Goues, N. Holtschulte, E. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *ACM Transactions on Software Engineering*, 41(12):1236–1256, 2015.

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

[47] Li Yujian and Xu Liye. Unweighted multiple group method with arithmetic mean. In *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, pages 830–834, Sep. 2010.

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

[49] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 702–713, New York, NY, USA, 2016. ACM.

[50] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[51] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[52] M. Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, Jan. 2018.

[53] M. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, https://www.cl.cam.ac.uk/~mom22/thesis.pdf, 2009.

[54] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Sofware Engineering*, pages 772–781, 2013.

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

[56] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering*, pages 683–693, 2012.

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

[58] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to generate disjunctive invariants. In *International Conference on Software Engineering*, pages 608–619, 2014.

[59] T. Nguyen, W. Weimer, C. Le Goues, and S. Forrest. Using execution paths to evolve software patches. In *Workshop on Search-Based Software Testing*, 2009.

[60] A. Nguyen-Tuong, D. Melski, J. W. Davidson, W. Hawkins, J. D. Hiser, D. Morris, D. Nguyen, E. Rizzi, et al. Xandra: An autonomous cyber battle system for the cyber grand challenge. *IEEE Security & Privacy*, 16(2):42–51, 2018.

[61] Office of the Secretary of Defense. Unmanned aircraft systems roadmap, 2005–2030, https://www.hsdl.org/?abstract&did=236553. Technical Report ADA445081, Washington DC, January 2005.

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

[63] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[64] C. Pellerin. DARPA goal for cybersecurity: Change the game. *DoD News*, December 2010.

[65] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, pages 87–102, 2009.

[66] D. Potts, R. Bourquin, L. Andresen, J. Andronick, G. Klein, and G. Heiser. Mathematically verified software kernels: Raising the bar for high assurance implementations. Technical report, General Dynamics, https://ts.data61.csiro.au/publications/nicta_full_text/8257.pdf, 2014.

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

[68] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, 2013.

[69] T. J. Ryan, G. M. Alarcon, C. Walter, R. F. Gamble, S. A. Jessup, A. A. Capiola, and M. D. Pfahler. Trust in automated software repair - the effects of repair source, transparency, and programmer experience on perceived trustworthiness and trust. In *HCI for Cybersecurity, Privacy and Trust - First International Conference, HCI-CPT 2019, Held as Part of the 21st*

HCI International Conference, HCII 2019, Orlando, FL, USA, July 26-31, 2019, Proceedings, pages 452–470, 2019.

[70] S. Saha, R. K. Saha, and M. R. Prasad. Harnessing evolution for multi-hunk program repair. *CoRR*, abs/1906.08903, 2019.

[71] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest. Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–328, 2013.

[72] E. Schulte, W. Weimer, and S. Forrest. Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair. In *The First Intl. Genetic Improvement Workshop (GI)*, 2015. Best paper award.

[73] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 532–543, Bergamo, Italy, September 2015.

[74] E. W. Smith. *Axe: An Automated Formal Equivalence Checking Tool for Programs*. PhD thesis, Stanford University, https://www.kestrel.edu/home/people/eric.smith/dissertation.pdf, 2011.

[75] E. W. Smith. Using Axe to reason about binary code (rump session). In *Proceedings of the Fourteenth International Workshop on the ACL2 Theorem Prover and its Applications*, May 2017.

[76] T. I. Sookoor, T. W. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: global views of distributed program execution. In *International Conference on Embedded Networked Sensor Systems*, pages 141–154, 2009.

[77] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In *Mining Software Repositories*, pages 512–515, 2016.

[78] C. S. Timperley. *Advanced Techniques for Search-Based Program Repair*. PhD thesis, University of York, http://etheses.whiterose.ac.uk/18466/, June 2017.

[79] C. S. Timperley, S. Stepney, and C. Le Goues. Poster: BugZoo: A Platform for Studying Software Bugs. In *International Conference on Software Engineering: Companion Proceedings*, ICSE Poster '18, pages 446–447. ACM, May 2018.

[80] W. Weimer. Advances in automated program repair and a call to arms. In *International Symposium on Search Based Software Engineering*, pages 1–3, 2013.

[81] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116, 2010.

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

[83] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

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

[85] Y. Yuan and W. Banzhaf. ARJA: automated repair of java programs via multi-objective genetic programming. *CoRR*, abs/1712.07804, 2017.

List of Symbols, Abbreviations and Acronyms

ACL2 A Computational Logic for Applicative Common Lisp — a theorem prover used by the Kestrel team

AES Advanced Encryption Standard — a block cipher specification for encrypting electronic data

AFL American Fuzzy Lop — a binary fuzz testing tool

AFRL Air Force Research Laboratories

API Application Programming Interface — a set of functions and procedures for the creation of applications that access the features or data of an operating system, application, or other service

ARJA Automated Repair of Java Programs — an algorithm for repairing Java programs using multi-objective genetic programming

ARM Advanced RISC Machine — a family of reduced instruction set computing architectures often used in embedded systems

BILR Block-level Instruction Location Randomization — Helix transformation to diversify code layout implemented using Zipr

BinART Binary Auto-Repair Templates — Helix transformation for common vulnerability patterns

C2 Command and Control — in this setting, a software system empowering designated personnel to exercise lawful authority and direction over an autonomous vehicle

CFAR Cyber Fault-tolerant Attack Recovery — DARPA program focusing on defensive cyber techniques

CFG Control-Flow Graph — a useful abstract representation of a program

CFI Control-Flow Integrity — defeats certain classes of security attacks

CGC Cyber Grand Challenge — a DARPA competition to create automatic defensive systems capable of reasoning about flaws, formulating patches and deploying them on a network in real time

CMT Continuous Measurement of Trust — a defensive technology for software

COTS Commercial off-the-shelf — generic packaged solutions which can be adapted to satisfy specific needs

CPS Cyber-physical system

CPU Central Processing Unit

DARPA Defense Advanced Research Projects Agency

eBPF	Extended Berkeley Packet Filters — used by CMT to assess trust
EM	Electromagnetic — electric-, magnetic-, radio- or radiation-based
GCS	Ground Control System
GenProg	Generic Approach to Automated Program Repair — a defensive technology for software
GLX	Global-layout Transformation — Helix transformation to relocate global variables
GPS	Global Positioning System
HACMS	High Assurance Cyber Military Systems — DARPA program to create technology for the construction of high-assurance cyber-physical systems
Helix	Name of binary transformation platform
HIL	Hardware in the Loop — a simulation that includes electrical emulation of sensors and actuators
HLX	Heap-layout Transformation — Helix transformation to modify heap layout
I/O	Input / Output — computer system communications with an outside environment
IoT	Internet of Things — a system of interrelated computing devices, mechanical and digital machines and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction
IMU	Inertial measurement units — an electronic device that measures and reports a body's specific force, angular rate, and sometimes the orientation of the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers
IRDB	Intermediate Representation Database — Helix storage and abstraction for representing and accessing information about binaries
JVM	Java Virtual Machine — a virtual machine that enables a computer to run Java programs
LR	Link Register — a special-purpose register which holds the address to return to when a function call completes
M-AIR	University of Michigan M-Air — an outdoor netted scientific facility for operating autonomous vehicles
MavLink	Micro Autonomous Vehicle Link Protocol — used by commodity autonomous vehicles
MAVROS	Micro Autonomous Vehicle Link Robot Operating System — a MAVLink extendable communication node for ROS
MD-5	Message Digest Algorithm 5 — a hashing, checksum and message digest algorithm often considered in cryptographic settings

MIPS Microprocessor without Interlocked Pipelined Stages — an influential reduced instruction set computer architecture

MURI Multidisciplinary University Research Initiative — a tri-service Department of Defense program that supports research teams

MUSE Mining and Understanding System Enclaves — a DARPA program that seeks to make significant advances in the way software is built, debugged, verified, maintained and understood

OS Operating System

QEMU Quick Emulator — free and open-source emulator that performs hardware virtualization

ROP Return-oriented Programming — a security attack

ROS Robot Operating System — middleware used by many autonomous vehicles

SBFL Spectrum-Based Fault Localization — localizes defects in programs

SBIR Small Business Innovation Research — a program coordinated by the Small Business Administration intended to help certain small businesses conduct research and development

SCFI Selective Control-Flow Integrity — Helix transformation for enforcing control-flow integrity

SDK Software Development Kit — collection of software development tools, often for interfacing with a particular project or library

SHA-1 Secure Hash Algorithm 1 — a cryptographic hash function that produces message digests associated with electronic data

SiTL Software in the Loop

SLX Stack-layout Transformation — Helix transformation to add canary and random padding to stack frames

SMT Satisfiability Modulo Theories — constraints used in formal verification

SPEC Standard Performance Evaluation Corporation — a series of benchmarks used to evaluate computer performance

SQL Structured Query Language — a common interface for database systems and transactions

STARS STARS Binary Analysis Engine — handles function recovery, exception handling, object layout, control and dataflow graphs

START Software Techniques for Automated Resilience and Trust

TC Turing Complete — a system that is computationally universal and can simulate any Turing machine

TCP Transmission Control Protocol — a reliable networking protocol associated with the Internet

TRMO Trusted and Resilient Mission Operation — the overall software defensive framework described in this report

TRSYS Trusted and Resilient Software-intensive Systems — a technical area of the Air Force Foundations of Trusted Computational Information Systems program, focusing on developing techniques to guarantee trust and maintain this trust as self-healing/resilient techniques fight through cyber-attacks

UAV Unmanned Aerial Vehicle

UDP User Datagram Protocol — a less-reliable networking protocol associated with the Internet

UML Unified Modeling Language — a general-purpose standard way to visualize the design of a system

UPGMA Unweighted Pair Group Method with Arithmetic Mean — an agglomerative (bottom-up) hierarchical clustering method

V&V Verification and Validation — procedures that are used together for checking that a product, service, or system meets requirements and specifications and that it fulfills its intended purpose

XML Extensible Markup Language — a markup language for encoding documents in a format that is both human-readable and machine-readable

ZAFL Zipr-based AFL — Fast binary fuzzer using Helix toolchain

Zipr Name of static rewriter used in the Helix toolchain. BILR is implemented as a Zipr plugin