AFRL-RI-RS-TR-2018-182

COOPERATIVE, TRUSTED SOFTWARE REPAIR FOR CYBER PHYSICAL SYSTEM RESILIENCY

UNIVERSITY OF VIRGINIA

JULY 2018

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

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

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2018-182 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
PATRICK M. HURLEY
Work Unit Manager

/ S /
JOSEPH A. CAROLI
Acting Technical Advisor
Computing & Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* JUL 2018	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* FEB 2015 – JAN 2018

4. TITLE AND SUBTITLE COOPERATIVE, TRUSTED SOFTWARE REPAIR FOR CYBER PHYSICAL SYSTEM RESILIENCY	**5a. CONTRACT NUMBER** FA8750-15-2-0075
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) Westley Weimer, Stephanie Forrest, Claire Le Goues, Miryung Kim	**5d. PROJECT NUMBER** T2RS
	5e. TASK NUMBER R1
	5f. WORK UNIT NUMBER KY
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Virginia, 1001 N. Emmet St., Charlottesville, VA 22903-4833	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2018-182

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

13. SUPPLEMENTARY NOTES

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

15. SUBJECT TERMS
Trusted and Resilient System, Automated SW Repair, Repair Quality, Repair Correctness, Trust Evidence, Runtime Verification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** PATRICK M. HURLEY
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	18	**19b. TELEPHONE NUMBER** *(Include area code)*

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

Table of Contents

1 Summary

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

2 Introduction

At a high level, for our work in Trusted Software Repair we:

- Developed a new methodology for CPS repair (i.e., improved algorithms)
  - We leveraged and combined advances in the detection and application of systematic edits, as well as in the automated repair of general software defects.

  - We investigated algorithms and hardware architectures to apply repairs to autonomous vehicles.

- Developed techniques that increase trust in a repaired system (i.e., improved trust)

  - We developed lightweight techniques, including the simulation and visualization of repairs or statistical evidence of repair quality. We initially focused on the simulation of repairs through the construction of specialized test inputs that highlighted the impact of a repair, as well as measuring and predicting trustworthiness of repairs.

  - We developed formal techniques, including the application of formal specifications and the production of formal proofs associated with repairs. We initially focused on a constructive reduction between program repair and program reachability, ultimately allowing repair proofs to be produced by way of proof-generating reachability analyses.

3 Methods, Assumptions, and Procedures

Over the course of this project, we developed techniques to improve software resilience using program repair. Program repair techniques operate on the source or binary code of a program and synthesize repair actions to address, fight through or otherwise overcome defects. They typically operate by considering changes to the program, searching through such changes or solving constraints to find them. Desired changes both defeat the bug in question and also retain all required functionality, which is typically expressed by test cases, pre- and post-conditions, or other formal invariants.

Our approach synthesizes repairs via software mutation. This can provide significant resilience in the face of security attacks and latent software engineering defects. However, the effect and merit of such repair actions may not be obvious to the human operators who must ultimately make deployment decisions. As a result, supporting trust in systems that make use of automated program repair is an ongoing research question. To improve trust in software, we gathered multiple modalities of evidence (including statistical evidence, differential testing, and formal invariants). This evidence can be presented to human operators; at a high level it answers questions such as “in what ways is this system similar to systems I already trust?”, “how would this system behave if I tested it in a way similar to the way I test systems I already trust?”, and “how can I characterize the properties that will hold as this system executes?”.

In support of this project, we also developed a unified testing platform using software- and hardware-in-the-loop as well as a physical autonomous vehicle platform based on the Iris+ Pixhawk, a consumer-grade uncrewed aerial vehicle (UAV). This testing platform allowed us to evaluate our ideas on realistic software and hardware.

At a high level, our framework focus involves the development of new algorithms for automated program repair that target CPS applications by combining insights from mutation-based single-defect repair and also from the abstraction of systematic edits. In addition, we developed a program repair algorithm based on a constructive reduction between program synthesis and program reachability, since the nature of the reduction admits proof generation. In addition, we developed methods for presenting evidence to the user in support of trust in the system, considering three approaches to such evidence, that explore the tradeoff between ease of use and strength of provided guarantees.

3.1 Mission Support Platform

We tested and deployed both the Erle-Copter and Iris+ Pixhawk uncrewed aerial vehicle systems in both indoor flight trials and tethered outdoor flights at the University of Virignia's Milton Airfield. These vehicles use off-the-shelf open-source software such as

ArduPilot, ArduCopter and APMrover. As a result of this testing, the Iris+ Pixhawk emerged as a unified platform that we could use in tandem with our collaborators. We collaborated with Raytheon BBN, the University of New Mexico, and the University of California at Los Angeles in the construction and deployment of this experimental prototype.

3.2 Unified Defect Scenarios

To evaluate all of the components of our system (e.g., resilience through repair, trust through evidence) in an end-to-end manner, we focused on a single indicative scenario. This scenario included a simple mission (the quadcopter was instructed to visit a number of waypoints in sequence) and a latent software bug or security attack that would prevent mission completion. Our goal was to fight through the software bug via our automated program repair approaches, complete the mission, and also provide evidence admitting operator trust in the post-repair system.

We considered two disruptions to the mission: a latent (non-malicious) software engineering defect, and a more active security attack. We describe the software bug first. We injected an indicative defect into the codebase: it causes the platform to only turn (yaw) in one direction under manual flight control. That is, if the operator uses a remote controller to rotate right, it would instead rotate left. Using this injected defect, we evaluated several techniques for measuring trust and resilience before and after observing the bug.

We then considered autonomous flight to demonstrate a repair scenario. We evaluated a situation in which a patrolling UAV encounters a defect, hovers in safe mode while constructing a repair, deploys the repair to complete the mission, and shares the abstract repair and trust-building evidence with users. We constructed a joint demonstration with BBN, augmenting our software-level notions of trust with their flight-telemetry-level notions of trust. We constructed flight plans based around Milton Airfield to demonstrate the techniques we developed during the period of performance. The flight control software automatically moves the UAV from waypoint to waypoint in sequence.

Our second mission disruption was a hostile software security attack. We introduced a controlled attacker that compromises the vehicle when it travels closely enough to a particular waypoint. We imagine a somewhat-remote attacker with a directional antennae; upon entering a “cone of influence” of the attacker, an unprotected vehicle would be subject to the control of the attacker. We proposed a model stealthy attacker who would not want to necessarily crash or control the vehicle, but rather cause the mission to be incomplete—e.g., to prevent it from photographing a point-of-interest near a particular waypoint. This scenario served as the basis for a large integrated demo involving UVA and BBN.

Ultimately, we sought to improve UAV mission resilience and trustworthiness by (1) learning a model of the intended mission in software and hardware simulation, (2) measuring flight telemetry during mission execution to quantify how trustworthy the UAV's behavior was in relation to the modeled mission, (3) if trust is violated, force the UAV to hover in-place (i.e., to pause) while we generate a candidate repair, (4) evaluate the quality of the repair via simulation, and (5) deploy the new patch to the UAV, at which point it can continue its mission.

3.3 Composable and Measurable Views of Trust

The BBN team, operating under a separate contract, focused on a model they call Composable and Measurable Views of Trust (CMT). Their approach provided a firm framework into which we integrated our lightweight and formal evidence of repair quality, as part of a whole-system trust and assurance case. In addition, the BBN team's proposal for hardware-in-the-loop simulation helped facilitate the development of an eventual unified experimental platform.

At a high level, BBN's CMT framework measures flight telemetry data and compares run-time statistics of those data against data expected based upon a trained model in simulation. A trusted mission is simulated beforehand, helping to establish a baseline of trustworthy telemetry data. At runtime after deployment in the field, the real flight telemetry data is gathered and compared against the trained model culminating in a quantifiable trust level—the more similar the runtime data is to the trained data, the more trust we have in the mission integrity. If the runtime data drifts too far from the modeled data, a trust violation occurs, which alerts us to begin repairing the flight software to reestablish trust. Additionally, after the repair is generated, the CMT framework helps reevaluate the quality of the repair in simulation. Finally, if the repair is of high enough quality, we deploy the repair to the UAV, at which point it can continue its original mission.

4 Results and Discussion

Our algorithm for automated software repair is known as GenProg (a generic program repair approach based on genetic programming). When an attack or defect is detected, it synthesizes repairs (changes to the software) to provide resiliency. It serves as both an indicative state-of-the-art technique and a baseline. We evaluated our improvements to it, along with evidence to promote trust, using the UAV scenario described above.

4.1 Tool Support for Repair and Statistical Evidence of Trust

The CMU team developed a framework and infrastructure for the GenProg-style repair of C++ programs. It was built atop LLVM’s Clang LibTooling library, a widely-used software infrastructure. Ultimately, GenProg is limited by its ability to rapidly localize faults. To address this limitation, we developed BigSift, a fault localization algorithm based on delta debugging. Using this technique, we achieved order of magnitude performance speedups of the fault localization component compared to default GenProg. In addition, we were able to improve the precision of fault localization by multiple orders of magnitude compared to Apache Spark’s Titian, an off-the-shelf solution. Finally, BigSift was able to localize fault-inducing input data within 62% of the original mission running time.
We also developed supervised models that use dynamic binary runtime signals to predict whether a program displays the correct behavior on a given input. Our insight, akin to anomaly intrusion detection, is that a collection of measurements regarding runtime behavior (e.g., maximum program counter values, number of branches taken, etc.) can help characterize correct and incorrect execution status. The goal is to develop continuous measures of program correctness to augment traditional discrete measures (e.g., passing or failing a test, having a proof or not). In our experiments, results for decision tree models based on instrumented binaries (using Intel's Pin instrumentation architecture) are promising (e.g., in terms of metrics such as precision, recall, accuracy and F-measure). This sort of runtime monitoring admits easy integration with the BBN CMT approach.

In practice, we found that employing CMT using a hardware-in-the-loop simulation was more indicative of real mission data than pure software simulation. Indeed, this approach worked well as a means of evaluating repair quality.

4.2 Measuring Trust via Differential Testing

Code clones are a common pattern of similarity in software. When applying edits to clones, developers often find it difficult to ensure the correctness of similar edits. Existing techniques check syntactic consistency but do not help examine the behavioral differences between clones. The problem is exacerbated when some clones are tested while their counterparts are not. Since new patches are typically untested, we believe that an operator's trust in a system could be improved by augmenting an edit (i.e., a patch or repair) with high-quality tests. To address this issue, we developed Grafter, a differential testing technique to identify behavioral differences between clones via code transplantation. Informally, Grafter allows one to evaluate a new piece of code (a patch that provides resilience) by adapting (grafting) previously-developed test cases.

To reuse the same test on similar code, Grafter adapts one clone to the counterpart clone by (1) identifying variations in identifier names, types, and method call targets between clones, (2) resolving compilation errors caused by such variations using code transformation rules, and (3) inserting stub code to transfer input data and intermediate output values for examination. In our experiments, Grafter successfully reused tests in 17 pairs of clones from the Apache Ant project, a large indicative software codebase, without inducing build errors, demonstrating automated transplantation capability. Grafter is robust at detecting faults seeded by a mutation testing tool, Major, a state-of-the-art approach for simulating indicative software defects. While a static clone bug finding tool detects 46% of faults only, Grafter detects 85% at the state-comparison level and 52% at the test-comparison level. Grafter reports behavioral divergence caused by calling different methods or using different constants, which previous static clone analysis tools fail to identify. This shows that Grafter can effectively complement existing static techniques in examining the runtime behavior of clones. In this context, an examination into runtime behavior via a new test corresponds to a new modality of evidence to increase operator trust in a post-repair system.

This approach was directly evaluated on the GenProg-generated repairs for the unified defect scenario. We first consider the source code location of the repairs in the APMrover2 codebase. Using our code detection analyses, we found similar software methods in the ArduCopter, ArduPlane and ArduSub codebases, which correspond to quadrotor helicopter, fixed-wing aircraft and underwater vehicles, respectively. These different codebases have different test suites. While statement coverage (i.e., the percentage of statements of code that are executed by a test suite) is only 23% for the entire system, 46% of clone groups have at least one tested clone, suggesting that applying our method can improve trust. After identifying the similar methods in the three other code bases, we first transplanted the defect to each of those similar methods. We then transplanted each of the GenProg-generated repairs on top of each transplanted defect. The final step was to evaluate each patched buggy clone with respect to its original test suite. Our hypothesis is that repairs that pass more tests in these transplanted settings are more likely to be of high quality, and that information about passing such tests can be presented as trust evidence to the user.

On the unified defect scenario, this approach found that five of the twelve GenProg patches passed all transplanted tests, and thus ranked those five patches as the most trustworthy. Independently, we had humans carefully evaluate the produced patches. Those human operators designated four of the candidate patches as most “correct”. All four of the “correct” patches were included in that group of five. Thus, our approach placed all of the highest-quality patches at the top of the ranking, supporting operator trust.

4.3 Formal Proofs, Repairs, and Trust

We have experimented with, and integrated, formal theorem-proving and constraint-solving tools such as Frama-C, Why3, Z3 and CVC4 with GenProg to study the repairs GenProg generates. In addition, we were interested in what we can learn about the formal specification of a program given its source code and test suite. In particular, we were interested in developing a metric for measuring populations of neutral variants of a program.

We conducted evaluations on two small, separate programs: egcd (greatest common divisor) and sqrt_lcm (square root of the least common multiple). Given a buggy version of a program we first repaired it with (unmodified) GenProg. We then generated 100 neutral variants of the repair, each containing five edits. A neutral variant, from evolutionary biology, is an individual with a different genotype but the same fitness or relevant phenotypical behavior. In software, a neutral variant is a modified version of the program that still passes all of the original test cases: it may be an alternate implementation of a key algorithm, for example. Neutral variants are useful because they represent diversity and a shifting attack surface, while maintaining required mission functionality for end users.

These neutral variants corresponded to alternate repairs that may have different formal properties and thus may be easier or harder for an operator to trust. We then used our dynamic invariant detection algorithms to infer invariants automatically in each alternate repair. Finally, we compared and contrasted the invariants found in the buggy program with those present in the various alternate repairs.

In both cases (egcs and sqrt_lcm) there was an invariant present in the repaired version of the program that was not present in the buggy version. In these case studies, the invariant that describes the repair was found in 100% of the sqrt_lcm alternate repairs and 99% of the egcd alternate repairs. We believe that the automatic identification of an invariant that corresponds to the defect could form the basis for a stronger algorithm to increase trust in post-repair systems. Most directly, a formal proof that an alternate repair maintains the invariant that is statistically most common to solutions to the defect may be of use. More abstractly, it may be possible to learn the invariant that all/most GenProg repairs have in common and then re-synthesize a repair based just on that invariant.

These shared invariants also have the potential to provide information about test suites and corner cases. This is related to another of our proposed evidence modalities for repair trust: test cases or simulation runs that highlight pre- and post-repair system behavior to help explain the effect of a repair. In the sqrt_lcm case study, while all of the alternate repairs shared the same “repair the bug” invariant, only 74% of them shared another key invariant that describes a corner case correctness issue of the sqrt_lcm function. Upon investigation, we found that the 26% of alternate repairs missing that invariant would have been eliminated if the sqrt_lcm scenario had included a stronger test case. That is, the repair process was unintentionally given too much freedom because of a weak test suite, and that was discovered via an analysis of which formal invariants were maintained. Similarly, the 1% of egcd alternate repairs that did not have “repair the bug” invariant was not a desired fix, and a test case demonstrating that can easily be constructed using our differential testing techniques.

We also studied a second important class of invariants. Our algorithms can use loop invariants to differentiate among programs that use different algorithms to perform calculations. We evaluated how well our method distinguishes between two different implementations of the same specified program. We used strong test suites to generate 50 neutral mutants of one implementation and 50 neutral mutants of a different implementation of the same specification for each of 5 different mathematical programs: egcd, intdiv, lcm, prod, and sqrt. For each program, we considered the generated variants into a single group of size 100. We then used our algorithm to partition those mutants by loop invariants into a partially ordered set of equivalence classes. The results showed that mutants generated from the first implementation were partitioned into disjoint equivalence classes from mutants generated from the second implementation, for four out of five of the programs. That is, a formal methods approach based on loop invariants allowed us to distinguish between diverse implementations.

4.4 Program Repair via Reachability

We developed a constructive polynomial-time reduction between template-based program repair (e.g., GenProg, a program synthesis task that provides resiliency) and program reachability (such as the already-existing software maintenance activities of software model checking or test input generation). A formal reduction allows one activity to be carried out in terms of another. That is, we developed a way to produce repairs using previously-existing, mature, and optimized maintenance reachability tools (and vice-versa).

The reduction is constructive in the sense of constructive logic: it provides a concrete algorithm for converting instances of one problem to another. The primary insight is that correctness constraints (i.e., test cases or pre- and post-conditions) in template-based program synthesis can be encoded as conditional guards (i.e., path predicates) in program reachability and vice-versa. The secondary insight is that unknown template values in template- based program synthesis can be encoded as input variables in program reachability and vice-versa.

We have developed, implemented, evaluated, and proved correct a prototype version of this algorithm. Its performance was comparable to that of existing repair algorithms such as AE, SemFix and previous versions of GenProg. This new algorithm has two implications. First, template-based program synthesis instances (i.e., program repair problems) can be converted and passed to black-box reachability solvers (e.g., software model checkers such as SLAM or BLAST or CVC, or reachability tools such as KLEE), which are generally more mature and efficient. Second, and more relevant for this project, this algorithm allows for the generation of proofs associated with program repair, through the use of a proof-generating reachability tool.

4.5 Hardware-Assisted Monitoring of Untrusted Systems

We proposed, and evaluated, a “dual-controller” architecture for resilience and trust in autonomous vehicle operation. In this architecture, an untrusted locomotion controller is more visible on the network and is responsible for mission and payload elements such as visiting waypoints and taking and analyzing pictures. The untrusted locomotion controller has high functionality, and may be constructed from the latest off-the-shelf and bespoke components, but is thus more vulnerable to attack. In addition, a trusted repair controller is responsible only for trust assessment, intrusion detection, repair construction, and safely hovering. It involves a much smaller code base, is not visible to an outside network, and can assume exclusive control of the sensors and actuators in the quadcopter. The trusted repair controller becomes part of the trusted code base. Because it stands apart from, and can take control of, the untrusted locomotion controller, it can fight through attacks on the locomotion controller via software repair.

Our dual-controller architecture thus involves one autonomous vehicle platform that features an untrusted, public-facing locomotion controller and a trusted repair controller. The trusted repair controller can monitor and reflash the locomotion controller, but not vice-versa. To help guarantee isolation, we implemented the locomotion and repair controllers on separate CPUs.

Techniques were investigated for implementing the trusted repair controller and untrusted locomotion controller on the same CPU using Intel’s Software Guard Extensions (SGX) and System Management Mode (SMM).

In this setup, the autonomous vehicle CPU can be viewed as running a virtual machine monitor or hypervisor: the locomotion software runs as a guest virtual machine and monitoring and accounting activities run outside the guest. However, certain security exploits can allow a malicious or compromised guest to either escape or subvert the hypervisor. In addition, and perhaps more insidiously, attacks have been reported that allow a guest to carefully time certain behavior so as to fool higher-level monitoring, resource accounting, and similar anomaly intrusion detection. The former are called VM Escape Attacks and the latter are called Resource Interference Attacks.

We proposed Scotch, a system for transparent resource accounting that uses Intel System Management Mode to perform monitoring, relaying data to a Secure Guard Extensions (SGX) trusted enclave. SGX provides an enclave-based trusted execution environment, allowing code to run in isolation: hardware support provided by Intel prevents the untrusted code from seeing or tampering with the secure monitoring code. In evaluations, our approach was able to detect scheduler attacks (i.e., was not fooled by compromised systems that attempted to avoid anomaly intrusion detection). Our approach was also low overhead, adding about 1μs per context switch, compared to 7μs per context switch for Xen; ultimately this resulted in a 0.0033% system overhead on CPU-bound workloads.

5 Conclusions

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

First, we developed BigSift, a fault localization algorithm targeted for automated debugging and repair. BigSift improved fault localization performance over the state of the art. This dramatic performance increase has addressed a significant shortcoming in repair approaches like GenProg whose search space, and thus performance, depends directly on the ability to localize defects.

Second, we leveraged a commercially-available autonomous vehicle package to deploy a Hardware-in-the-Loop simulation capable of assessing repairs generated to address faulty software. Such a HIL simulation is more indicative than a pure-software simulation, which is incapable of simulating hardware-level idiosyncrasies. Further, the HIL simulation is integral in performing repairs during missions in deployment.

Third, we devised and proved a bidirectional polynomial time constructive reduction between the program-reachability problem and the template-based program synthesis problem. We took advantage of the insight that correctness constraints in template-based program synthesis can be represented as conditional guards in program reachability. We have developed, implemented, evaluated, and proved correct a prototype version of this algorithm. Its performance was comparable to that of existing repair algorithms, but it more easily admits trust.

Fourth, we developed supervised models that use dynamic binary runtime signals to predict whether a program exhibits correct behavior on a given input. Our insight, much like anomaly detection, is that a collection of measurements regarding runtime behavior (e.g., performance counters) can help characterize correct and incorrect execution status. We developed continuous measures of program correctness to augment traditional discrete measures. In practice, we found that employing this CMT approach using a hardware-in-the-loop simulation was more indicative of real mission data than pure software simulation. Indeed, this approach worked well as a means of evaluating repair quality.

Fifth, we developed an approach to differential testing of automatically-constructed repairs. Briefly, automatically-generated patches may be under-tested in that the test suite may not encompass all desirable behavior. By leveraging test suites from similar locations, we can gain trust in the automatically-generated patches. In our experiments, this test transplantation approach provides an increased test suite that ranks correct patches highly, facilitating operator trust.

Finally, we leverage formal invariants to determine semantic differences between defective and patched versions of rover software. We differentiate correct repairs that behave as intended from plausible repairs that pass the test suite but introduce undesirable behavior. In our experimentation, desirably-correct and merely-plausible patches were distinguishable.

In summary, we produced two live demonstrations showcasing our combination of techniques for providing resiliency and trust in autonomous vehicle missions. We developed a defect scenario in which a simulated attacker compromises and autonomous vehicle, and we automatically detect and repair the issue to fight through the attack.

Appendix A: Publications

Westley Weimer, Stephanie Forrest, Miryung Kim, Claire Le Goues, Patrick Hurley: Trusted Software Repair for System Resiliency: Dependable Systems and Networks (DSN 2016) Industrial Track

Kate Highnam, Kevin Angstadt, Kevin Leach, Westley Weimer, Aaron Paulos, Patrick Hurley: An Uncrewed Aerial Vehicle Attack Scenario and Trustworthy Repair Architecture: Dependable Systems and Networks (DSN 2016) Industrial Track

T. Le, D. Lo, C. Le Goues, and L. Grunske. A Learning-to-Rank Based Fault Localization Approach using Likely Invariants. ACM International Symposium on Software Testing and Analysis (ISSTA 2016)

K. Leach, C. Spensky, W. Weimer and F. Zhang. Towards Transparent Introspection. In Software Analysis, Evolution and Reengineering (SANER) 2016.

K. Leach, C. Spensky, L. Barnes, and W. Weimer. A MapReduce Framework to Improve Template Matching Uncertainty. In Big Data and Smart computing (BigComp) 2016.

M. Moses, G. Bezerra, B. Edwards, J H. Brown, S. Forrest. Energy and Time Determine Scaling in Biological and Computer Designs. Philosophical Transactions of the Royal Society B 2016.

R. van Tonder and C. Le Goues. 2016. Defending against the attack of the micro-clones. In Proceedings of the 24th International Conference on Program Comprehension (ICPC 2016).

Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, Miryung Kim: Automated debugging in data-intensive scalable computing. Symposium on Cloud Computing (SoCC) 2017

Christopher Steven Timperley, Susan Stepney, Claire Le Goues: An Investigation into the Use of Mutation Analysis for Automated Program Repair. Symposium on Search-Based Software Engineering (SSBSE) 2017

Kevin Leach, Fengwei Zhang, Westley Weimer: Scotch: Combining Software Guard Extensions and System Management Mode to Monitor Cloud Resource Usage. Research in Attacks, Instructions and Defenses (RAID) 2017

ThanhVu Nguyen, Deepak Kapur, Stephanie Forrest, Westley Weimer: Connecting Program Synthesis and Reachability: Automatic Program Repair using Test-Input Generation. Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2017

J. Dorn, J. Lacomis, W. Weimer, and S. Forrest. Scalable software energy reduction. Transactions on Software Engineering, (in press).

C. Le Goues, Y. Brun, S. Forrest, and W. Westley. Clarifications on the construction and use of the manybugs benchmark. Transactions on Software Engineering, 43(11):1089–1090, (2017).

J. Ericksen, M. Moses, and S. Forrest. Automatically evolving a general controller for robot swarms. IEEE Symposium on Artificial Life, 2017.

Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan and Miryung Kim. Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow. International Conference on Software Engineering (ICSE), 2018.

Elena L. Glassman, Tianyi Zhang, Björn Hartmann, Miryung Kim. Visualizing API Usage Examples at Scale. Conference on Human Factors in Computing Systems (CHI), 2018

Just, Rene, Franz Schweiggert, and Gregory M. Kapfhammer. "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler." Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on. IEEE, 2011.

List of Symbols, Abbreviations, and Acronyms

BLAST	Berkeley Lazy Abstraction Verification Tool
CPS	Cyber-Physical System
CMT	Composable and Measurable views of Trust
CVC	Cooperating Validity Checker
DIG	Dynamic Invariant Generator
GenProg	Generic approach to Program repair based on Genetic Programming
HIL	Hardware-in-the-Loop simulation
LASE	Learning and Applying Systematic Edits
MAC	Media Access Control
MAVLink	Micro Air Vehicle Link
UAV	Uncrewed Autonomous Vehicle
SLAM	Software, Languages, Abstraction, Model checking
SGX	Secure Guard Extensions
SIL/SITL	Software-in-the-loop simulation