Trusted Software Repair for System Resiliency

Westley Weimer
Univ. of Virginia
Charlottesville, VA
weimer@virginia.edu

Stephanie Forrest
Univ. of New Mexico
Albuquerque, NM
forrest@cs.unm.edu

Miryung Kim
Univ. of California
Los Angeles, CA
miryung@cs.ucla.edu

Claire Le Goues
Carnegie Mellon Univ.
Pittsburgh, PA
clegoues@cs.cmu.edu

Patrick Hurley
Air Force Research
Rome, NY
patrick.hurley@us.af.mil

***Abstract*—We describe ongoing work to increase trust in resilient software systems. Automated software repair techniques promise to increase system resiliency, allowing missions to continue in the face of software defects. While a number of program repair approaches have been proposed, the most scalable and applicable of those techniques can be the most difficult to trust. Using approximate solutions to the oracle problem, we consider three approaches by which trust can be re-established in a post-repair system. Each approach learns or infers a different form of partial model of correct behavior from pre-repair observations; post-repair systems are evaluated with respect to those models. We focus on partial oracles modeled from external execution signals, derived from similar code fragment behavior, and inferred from invariant relations over local variables. We believe these three approaches can provide an expanded assessment of trust in a repaired, resilient system.**

I. Introduction

There is increasing demand for systems that are both trusted (e.g., [13]) and resilient (e.g., [22]). For this paper, the term *dependability* measures how consistently a system successfully completes its assigned mission [1]. We use *trust* to refer to the human belief that the system is dependable; the ultimate decision to deploy a system or not is usually made by human operators. A *resilient* system is capable of safely recovering from or avoiding errors, attacks or environmental challenges to complete its original mission or a variation thereof [9]. We desire systems that can be both resilient and trusted in the face of unanticipated challenges, and that can be applied to off-the-shelf components.

As software systems have become more multi-functional, autonomous, and tightly coupled with actions in the physical world, resiliency has been widely recognized as a desirable property. One approach to achieving resiliency is to anticipate all possible problems or states that the system could encounter and devise pre-programmed responses. The DARPA High Assurance Cyber Military Systems (HACMS) [13] program, which uses automated tools to partially or fully formally synthesize control software for various uncrewed platforms, exemplifies this approach. This approach admits significant trust, as synthesized code provides correctness, safety, and security guarantees with respect to the given specification. However, the HACMS approach does not provide resiliency in the face of unanticipated challenges and it may be difficult to apply such formal approaches to existing systems.

Acknowledgment Of Support And Disclaimer: (a) Contractor acknowledges Government's support in the publication of this paper. This material is based upon work funded by AFRL, under AFRL Contract No. FA8750-15-2-0075. (b) Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL. Approved for Public Release, Unlimited Distribution: 88ABW-2016-1342, 3/21/2016.

We focus on a second approach to providing system resiliency: automated program repair. This approach synthesizes *repair* actions or software patches that can be applied to a system, allowing it to overcome errors or attacks. The repair action avoids the buggy or incorrect behavior while retaining required functionality. An example of the second approach is the family of program repair methods that have been developed recently. Program repair approaches have the advantage that they have demonstrated resiliency in the face of unknown attacks [15] and defects [14] and can apply to off-the-shelf legacy systems. However, a repair synthesized by a black-box technique (e.g., a search- or constraint-based system) can be difficult to trust.

We argue that operator trust in a resilient system, including trust in the dependable operation of the system after a repair action, is critical for real-world deployment [13], [22]. A recent NASA survey, for example, ranked understanding and readability as more important than functional correctness when making use of software [8]. We thus propose to augment resilient program repair with three assessments that allow the human operator to trust in the dependable post-repair behavior of the system. The operator may trust the post-repair system by observing that it behaves correctly on expected and unexpected inputs. Our high-level insight is thus that these assessments are just aspects of the more general *oracle problem* in software engineering (e.g., [19]) applied to unannotated legacy systems. In the oracle-comparator formal model of testing, the oracle provides the expected correct output for a given input. If we can partially determine the correct response to a given input, we can trust the post-repair system to the extent that it displays that response.

We focus on three partial oracles, each of which applies to off-the-shelf systems by assuming that previous behavior is largely correct even though formal oracle annotations are unavailable. Each of these three approaches thus features a training or analysis phase in which a model of correct behavior (i.e., an oracle) is constructed.

We believe these three assessments, based on three partial solutions to the oracle problem for legacy software, can support human trust in post-repair resilient systems.

II. System Design

We envision a system that provides resiliency to unannotated software systems via program repair and augments that resiliency with three assessments to support operator trust in post-repair system dependability. This short paper

outlines a research project with that explicit goal, involving the University of Virginia, University of New Mexico, Carnegie Mellon University, and the University of California at Los Angeles. We first describe an existing adaptive repair framework, GENPROG. We then describe three assessments that can be made to post-repair systems. We use two guiding principles: each assessment derives from a partial solution to the oracle problem and each assessment trains or models correct behavior from existing trusted, but unannotated, systems.

First, we consider oracles based on external execution signals observed at run-time. We hypothesize that lightweight metrics of running programs (e.g., instruction execution counts, branch taken counts, etc.) can be combined to partially characterize correct program behavior [16]. Conceptually akin to anomaly detection, this approach involves training a model from metrics collected during trusted runs. Metrics collected during post-repair runs can then be compared to that model. While not comprehensive, this partial oracle has the advantage of being quantitative and easy to gather.

Second, we consider oracles based on behavior in similar code fragments. Defects and vulnerabilities often deviate from standard behavior [5] (e.g., a program with a buffer overrun in one program location often correctly bounds-checks arrays used in other program locations). This problem is exacerbated by the prevalence of code clones (e.g., copy-paste code) in existing systems [24]. Given a repair to a particular piece of code, we can adapt trusted test inputs and test oracles from similar code locations to the post-patch location. This adaptation analysis corresponds to training a partial model. The new code is correct if its behavior on the transformed old inputs matches the transformed old outputs. This partial oracle has the advantage of leveraging existing developer expertise by reusing extant tests.

Third, we consider oracles based on inferred and validated program invariants. Informally, once a defect or vulnerability has been repair, the post-repair system is correct if its behavior is similar to the pre-repair system except when faced with the defect or attack. We capture this notion more formally via invariant inference, learning a model of invariant relationships between program variables on trusted inputs. The post-repair code is correct if it also satisfies those invariants. While more expensive to apply, this more partial oracle has the advantage of providing more formal guarantees.

A. Resiliency — Automated Program Repair

GENPROG is a popular automated program repair approach [15]. Given a program, a test suite that encodes desired behavior, and evidence of a defect, GENPROG samples the space of possible repairs until one is found that retains required functionality while repairing the bug. GENPROG has been evaluated on real-world, high-priority defects from programs totaling over five million lines of code and guarded by over ten thousand tests, and is typically able to repair about half of such defects [14].

Although effective in practice, GENPROG's search-based technique offers fewer guarantees about synthesized patches than repair approaches based on constraint solving (e.g., [18]). Thus, while GENPROG repairs have been validated against existing regression test suites and have been evaluated post-facto for maintainability [7], acceptability [11] and security concerns [15], the system itself does not provide assessments to support human trust [23].

B. Assessment — Dynamic Execution Signals

We have developed a set of quantitative metrics to assess a candidate repair, based on a partial-correctness oracle modeled on dynamic execution signals observed on trusted inputs. We have previously found that it is possible to combine multiple comparatively simple, static metrics to usefully approximate source code quality [16]. With respect to trusted software repairs, and taking inspiration from anomaly detection, we have extended this idea to signals taken from dynamic runtime information.

Our key insight is that a program that produces unintended behavior on one input often exhibits observable, inconsistent behavior on other inputs. Moreover, we can observe behavioral inconsistencies in the values of measurable binary-level execution signals (e.g., number of instructions executed, number of branches taken, etc.), which are language-independent. Because no single execution signal fully characterizes correct program behavior, we apply machine learning techniques to train predictive models from numerous runtime signals. This process takes as input collections of dynamic signals from the program running on trusted test cases to produce a model that, given a new execution (e.g., a new test input on the same program, a regression test run on a changed program, a test input on a different program for which no other testing information is available), predicts whether the program appears to have executed correctly on the corresponding test input. We have had success using both supervised learning and also an unsupervised approach based on custom clustering [10].

We anticipate extending this approach to include both source-level and static information, such as similarity between a changed region of code and previous changes to that same code base, continuing to combine a variety of (possibly noisy) similarity metrics. This will enable the model to more directly relate the new source code and behavior to previous models of "known good" source code and behavior.

While not as formal as a proof, such statistical reasoning is easier to apply to off-the-shelf programs (e.g., it applies to binaries) and may be easier for users to understand. We can cast this statistical evidence technique as a problem in the domain of information retrieval, measuring the precision and recall (i.e., effectively measuring false positives and false negatives) between the ground truth set of high-quality program or repairs and the set of repairs endorsed by our statistics, as a function of the threshold of evidence chosen. This formulation produces a standard receiver operating characteristic curve evaluation.

C. Assessment — Targeted Differential Testing

We are developing techniques to adapt existing tests to target repaired code, based on partial-correctness oracles transformed

from similar program locations. These adapted tests assess pre- and post-repair behavior differences in relevant corner cases. There are many existing constraint-based test input generation techniques (e.g., [2], [6]); however, these techniques focus on input generation only, not the generation of expected test oracles, and are therefore insufficient for our task.

We are currently investigating a technique that can transplant existing tests to a new similar location in order to reuse the test inputs and corresponding oracle to perform differential testing. This approach is based on the insight that reusing code fragments via copying and pasting with minor edits is a common activity in software development [12] and that a significant fraction (between 7% to 23%) of code in large software systems consists of clones [24] (i.e., highly similar source code, as in copy-paste code).

We can analyze pre-repair behavior on code locations that are similar to the repaired location. Test inputs and their corresponding expected outputs are carved out and transplanted from one program location to another. In this model, the post-repair behavior is correct if it produces the transformed expected output when applied to the transformed input (this is called differential testing). In addition, this approach can also be used to compare the post-repair behavior of two similar code fragments. To do so, our test carving and transplantation technique exercises counterpart clones using the same test and compares the corresponding runtime behaviors.

This problem of reusing and adapting tests is challenging due to variations in program source structure (e.g., variable names, types, etc.). Such variations must be handled to construct well-typed tests and avoid incompatibilities in referenced objects. For example, if two clones use different types of objects, developers must currently manually construct coercions from one type to another. This approach of adapting existing tests may thus not always apply to unannotated code (e.g., if no clones are available for a given patch or if manual coercions must be provided). However, when it applies it has the advantage of leveraging existing developed expertise (from the input trusted tests) and directly highlighting post-repair system behavior in relevant expected and corner cases.

D. Assessment — Invariants and Proofs

We are developing techniques to formally prove certain aspects of post-repair system behavior, based on partial-correctness oracles that take the form of invariants over program variables from trusted runs. These invariants formally capture the intuitive notion that the post-repair system should behave similarly to the trusted pre-repair system (except with respect to the defect or vulnerability) by specifying which aspects of behavior must be maintained for correctness.

Our first step is to infer relevant invariants automatically; we do this by analyzing dynamic variable values from trusted runs of the pre-repair system. We have developed techniques for learning polynomial, non-linear, array-based, and disjunctive invariants [20], [21]. Although not sufficient for all software, this rich language of invariants can describe a number of partial specifications. Our technique, called DIG, provides a novel method for discovering these relations. In one test, it discovered 60% of the documented invariants necessary to fully formally verify the functional correctness of AES encryption [20] — i.e., it removes 60% of the manual annotation burden. Dynamic invariant detection techniques sometimes produce false positives (e.g., spurious conditions that only appear invariant because of limited testing); we retain only those invariants that can be formally proved to hold in the program (e.g., via k-induction) [20]. To summarize, our approach analyzes sample program traces to infer a set of candidate invariants that hold for the traces. It then uses a theorem prover to retain invariants that hold statically. Those invariants become the partial correctness oracle: the post-repair system must provably satisfy those invariants.

For each discovered invariant, we use verification condition generation from axiomatic semantics to generate a formula encoding the post-repair system's maintenance of the invariant. For example, if the pre-repair system is modeled as a method f and the post-repair system is g, full formal equivalence corresponds to $\forall x.\ f(x) = g(x)$. Full formal equivalence is both difficult to prove and undesirable because we want the repaired system to behave differently on at least the bug-inducing input. Instead, given an invariant I, we prove $(\forall x.\ I(f(x))) \implies (\forall x.\ I(g(x)))$. That is, given that the invariant holds for the original program, we seek to prove that it holds for the post-repair system. Machine-checkable proofs of such formulas can be constructed via automated theorem proving [3] or interactive proof assistants. In practice, repair actions change only small portions of an entire system [15]; the closer f is to g the more likely that the invariant held pre-repair will provably hold post-repair. Thus, even though such formal methods are relatively difficult to apply to off-the-shelf systems, we remain optimistic. When applicable, this assessment includes formal static reasoning about the correctness of the post-repair system.

III. Current Status and Results

a) Dynamic execution signals: The partial oracles derived from execution signals can characterize and predict correctness in a variety of scenarios, depending on the data used to construct the models. A model trained on a single program and existing test suite can predict whether the program performs correctly on new, generated test inputs (augmenting generated inputs with predicted oracles, and improving testing coverage). A model trained on multiple previous versions of a program with test cases can predict whether a modified version of the program (such as one that has been repaired in response to a vulnerability) is continuing to function correctly. A model trained on multiple existing programs can leverage data from existing projects to estimate correctness of a previously unseen, undertested program.

Our prototypes [10], which use Pin [17] to collect signals, provided promising results on a range of Unix utilities and embedded programs. The unsupervised models showed high precision and recall when evaluated on programs and test executions, with F-measure above 0.80 on 14/17 programs.

Augmenting an existing test suite by predicting oracles for generated inputs using a supervised learning technique showed comparable results; cross-program models predicted correctly more than half the time.

b) Targeted differential testing: We have developed GRAFTER, an implementation of our partial oracle approach based on differential testing. It takes similar program locations (clones) as input and automatically adapts and reuses the test(s) of one clone on its counterpart clone in three phases. GRAFTER first identifies the scope and extent of the code to be adapted, including the definition of referenced variables and methods. It then ports this code to the target location by leveraging multiple transformations to handle code variations. Finally, it synthesizes and inserts stub code to propagate input data to ported variables and then transfers intermediate output of grafted code back for examination. By implanting one clone in the place of another, GRAFTER enables assessment by reusing the same test on both clones without adapting the test code itself. In addition, compared with existing differential testing frameworks (e.g., [4]) which often check only whether the program behavior is the same for the pre-repair and post-repair versions, GRAFTER does not require the type, function, or object names to be the same across clones.

c) Invariants and proofs: We have already evaluated DIG, our tool for inferring polynomial, non-linear, array-based, and disjunctive invariants, on nonlinear arithmetic benchmarks and AES encryption [20], [21]. More recently, we have made a preliminary investigation of the use of invariants as an assessment for program repair. We considered an implementation of the extended GCD algorithm with a seeded defect. DIG inferred three relevant nonlinear invariants from correct execution on trusted input ($kx - iy + B = 0$, $jk - im + 1 = 0$ and $mx - jy - A = 0$). We then considered ten candidate repairs synthesized by GENPROG, eight of which were correct and two of which contained or introduced latent defects. We were able to prove, using Z3 [3], that all three invariants are maintained by the eight correct post-repair system. By contrast, the two incorrect repairs and the version with the seeded defect all failed to maintain two of the three invariants. While an evaluate on a single program is very preliminary, it does give confidence that we can infer invariants from trusted input and prove that they are maintained by a post-repair system, providing a relative assessment of candidate repairs.

IV. CONCLUSIONS

There is increasing demand for systems that are trusted, resilient, and apply to legacy systems. We propose to address this problem via automated program repair (for resiliency) augmented with assessments (for trust). We observe that determining correct behavior relates to the oracle problem, and propose three assessments based on partial oracles, each based on training or analyzing trusted data. We believe these approaches can provide an expanded assessment of trust in a repaired, resilient system.

REFERENCES

[1] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[2] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Operating Systems Design and Implementation*, 2008, pp. 209–224.

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

[4] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Foundations of Software Engineering*, 2006, pp. 253–264.

[5] D. R. Engler, D. Y. Chen, and A. Chou, "Bugs as inconsistent behavior: A general approach to inferring errors in systems code," in *Symposium on Operating Systems Principles*, 2001.

[6] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *International Symposium on Software Testing and Analysis*, 2011, pp. 364–374.

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

[8] N. S. R. W. Group, "Software reuse survey," in *http://www.esdswg.com/softwarereuse/Resources/library/working_group_documents/survey2005*, 2005.

[9] Y. Y. Haimes, "On the definition of resilience in systems," *Risk Analysis*, vol. 29, no. 4, pp. 498–501, November 2009.

[10] D. Katz, C. Camilo Jr., and C. Le Goues, "Using runtime behavioral signals to predict whether programs behave as intended," Carnegie Mellon, Tech. Rep., 2016.

[11] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *International Conference on Sofware Engineering*, 2013.

[12] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Empirical Software Engineering*, 2004, pp. 83–92.

[13] J. Launchbury, "High-assurance cyber military systems (HACMS)," *DARPA Program Information*, November 2015.

[14] 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.

[15] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automated software repair," *Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[16] C. Le Goues and W. Weimer, "Measuring code quality to improve specification mining," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 175–190, 2012.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Programming Language Design and Implementation*, 2005, pp. 190–200.

[18] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *International Conference on Software Engineering*, 2016 (To appear).

[19] C. D. Nguyen, A. Marchetto, and P. Tonella, "Automated oracles: An empirical study on cost and effectiveness," in *Foundations of Software Engineering*, 2013, pp. 136–146.

[20] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Dig: A dynamic invariant generator for polynomial and array invariants," *Transactions on Engineering and Methodology*, vol. 23, no. 4, 2014.

[21] ——, "Using dynamic analysis to discover polynomial and array invariants," in *International Conference on Software Engineering*, 2012, pp. 683–693.

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

[23] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[24] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.