Trusted Software Repair for System Resiliency

(future work in this award)
Westley Weimer, Stephanie Forrest, Miryung Kim, Claire Le Goues
Flight Control Software

• This demo's focus is on repairing flight data

• However, flight control software can contain security vulnerabilities as well as standard software engineering bugs
  • No DO-187B or ISO-26262 for the flight software used in the demo, etc. (cf. COTS, SOUP)
  • Version control logs reveal a striking number of bug fixes over time

• Subsequent demonstrations: source code
Automated Program Repair

• Any of a family of techniques that generate and validate or solve constraints to synthesize program patches or run-time changes
  • Typical Input: program (source or binary), notion of correctness (passing and failing tests)

• Program repair provides resiliency
  • Powerful enough to repair serious issues like Heartbleed, format string, buffer overruns, etc.

• Efficient (dollars per fix via cloud computing)
Program Repair Quality

- GenProg '09

Abstract

Automatic program repair has been a longstanding goal in software engineering, yet debugging remains a largely manual process. We introduce a fully automated method for locating and repairing bugs in software. The approach works on off-the-shelf legacy applications and does not require formal specifications, program annotations or special coding practices. Once a program fault is discovered, an extended form of genetic programming is used to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. Standard test cases are used to exercise the fault and to encode program requirements. After a successful repair has been discovered, it is minimized using structural differencing algorithms and delta debugging. We describe the proposed method and report experimental results demonstrating that it can successfully repair ten different C programs totaling 65,000 lines in under 200 seconds, on average.

To alleviate this burden, we propose an automatic technique for repairing program defects. Our approach does not require difficult formal specifications, program annotations or special coding practices. Instead, it works on off-the-shelf legacy applications and readily-available test cases. We use genetic programming to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. Our technique takes as input a program, a set of successful positive test cases that encode required program behavior, and a failing negative test case that demonstrates a defect.

Genetic programming (GP) is a computational method inspired by biological evolution, which discovers computer programs tailored to a particular task [19]. GP maintains a population of individual programs. Computational analogs of biological mutation and crossover produce program variants. Each variant’s suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. GP has solved an impressive range of problems (e.g., in [16]), but even beyond that it has new
Program Repair Quality

- GenProg '09 - minimize
- Remove spurious insertions
Program Repair Quality

- GenProg '09 - minimize
- PAR '13 - human changes
  - Mutation operations based on historical human edits
Program Repair Quality

- GenProg '09 - minimize
- PAR '13 - human changes
- Monperrus '14 - PAR is wrong
  - Experimental methodology has several issues
  - Patch prettiness is not patch quality
Program Repair Quality

- GenProg '09 - minimize
- PAR '13 - human changes
- Monperrus '14 - PAR is wrong
- SPR '15 - condition synthesis
  - Solve constraints to synthesize expressions for conditionals
  - Not just deletions
Program Repair Quality

- GenProg ’09 - minimize
- PAR ’13 - human changes
- Monperrus ’14 - PAR is wrong
- SPR ’15 - condition synthesis
- Angelix ’16 - SPR is wrong
  - SPR still deletes
  - Use semantics and synthesis

A recent study revealed that the majority of GenProg repairs avoid bugs simply by deleting functionality. We found that SPR, a state-of-the-art repair tool proposed in 2015, still deletes functionality in their many “plausible” repairs.
Resilient but Untrusted

- Program repair does provide resiliency
- But the “quality” of repairs is unclear
  - So they are not trusted
  - Thus far: algorithmic changes (e.g., mutation operators, condition synthesis, etc.)
- We are investigating a post hoc, repair-agnostic approach to increasing operator trust
  - Provide multiple modalities of evidence
  - Approximate solutions to the oracle problem
Trust Framework

- Augment repairs with three **assessments** that allow the human operator to trust in the post-repair dependable operation of the system
  - These assessments are aspects of the oracle problem for legacy systems
  - Each features a training or analysis phase in which a **model of correct behavior** (oracle) is constructed
Dynamic Execution Signals

- Insight: a program that produces unintended behavior for a given input often produces other observable inconsistent behavior
  - cf. printf debugging

- Measure binary execution signals
  - Number of instructions, number of branches, etc.

- In supervised learning, our models predict whether new program runs correspond to intended behavior quite accurately
Targeted Differential Testing

- Code clones (intentional or not) are prevalent
- Repairs are often under-tested
  - They may insert new code, etc.
- Insight: We can adapt tests designed for code clones to become tests targeted at repairs
  - Identify variants, transplant code, propagate data
- Successfully adapted tests in many examples
Invariants and Proofs

• Insight: The post-repair system is not equivalent to the pre-repair system, but it may maintain the same invariants (or more).

• Identify invariants, prove them correct
  • No spurious or incorrect invariants remain

• We can infer 60% of the documented invariants necessary to prove functional correctness of the Advanced Encryption Standard
  • Linear, nonlinear, disjunctive, and array invariants
Example: Zune Bug

- Ex. Invariants in Buggy Program
  - days_top > 365
- Ex. Correct Invariants
  - days_top > 365
  - days_bot < days_top
  - year_bot = year_top + 1

```c
void zunebug(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
    }
    printf("current year is %d\n", year);
}
```
Research Hypothesis

- Among test-equivalent program variants produced by mutation (e.g., among candidate repairs), those program variants that share common invariants respect program intent

- Why?
  - Exploits our duality between generate-and-validate program repair and mutation testing
  - “Mutation analysis” applied in reverse
  - Competent programmer hypothesis
Three-Phase Plan

• Given one candidate repair ...

• **Generate** a large number of neutral (or test-equivalent) alternate candidate repairs
  • Via a special directed neutral walk

• Dynamically infer and statically verify **invariants** of those candidate repairs

• Select repairs that respect majority invariants
Generating Alternate Repairs

- We can generate many neutral edits
  - Changes to a program that retain behavioral equivalence with respect to a test suite
  - But may behave differently for future attacks or unconsidered benign inputs
- Cheaply generate singleton neutral edits
- Then combine (or “cluster’) many of them to make a single candidate repair
  - But edits may depend on each other ...
- We use a directed neutral walk
Directed Neutral Walk

Neutral edits
Directed Neutral Walk

Neutral edits

Gather
Directed Neutral Walk

directed walk

\[
\text{random shuffle}
\]
Directed Neutral Walk

random shuffle

not neutral
Directed Neutral Walk

random shuffle

not neutral

neutral
Directed Neutral Walk

random shuffle

not neutral

neutral

iterate

recombination
Effective Combination

Combined Neutral Edits (out of 50)

Directed Neutral Walk

Baseline Recursive Selection
From Repair Candidates to Invariants

• We now have a large number of repair candidates
  • Each of which passes all test cases and contains a large number of neutral edits

• Next, we apply dynamic invariant generation
  • Record the values of variables on execution traces
  • Infer linear, non-linear polynomial, disjunctive and array invariants
  • Prove that each invariant holds (is not spurious)
Least Common Multiple program:

```c
int lcm(int a, int b)
    x = a; y = b; u = b; v = a;
    while (x != y)
        if (x > y)
            x=x-y; v=v+u;
        else
            y=y-x; u=u+v;
    return (u+v)/2;
```

Invariant Example
Invariant Example

Least Common Multiple program:

```c
int lcm(int a, int b) {
    x = a; y = b; u = b; v = a;
    while (x != y) {
        if (x > y) {
            x = x - y; v = v + u;
        } else {
            y = y - x; u = u + v;
        }
    }
    return (u + v) / 2;
}
```

Weak Test Suite:

lcm(1,1) = 1
Invariant Example

Least Common Multiple program:

```c
int lcm(int a, int b)
    x = a; y = b; u = b; v = a;
    while (x != y)
        if (x > y)
            x = x - y;  u = b;  v = v + u;
        else
            y = y - x;  u = u + v;
    return (u + v) / 2;
```

Weak Test Suite:

- lcm(1, 1) = 1
- Candidate
- Alternate
- Repair
Invariant Example

Least Common Multiple program:

```c
int lcm(int a, int b)
    x = a;  y = b;  u = b;  v = a;
    while (x != y)
        if (x > y)
            x=x-y;  v=v+u;
        else
            y=y-x;  u=u+v;
    return (u+v)/2;
```

Inferred Loop Invariant:

\[ u \times x + v \times y = 2 \times a \times b \]
Invariant Example

Least Common Multiple program:

```c
int lcm(int a, int b) {
  x = a; y = b; u = b; v = a;
  while (x != y) {
    if (x > y) {
      x = x - y; u = b; v = v + u;
    } else {
      y = y - x; u = u + v;
    }
  }
  return (u + v) / 2;
}
```

Weak Test Suite:

lcm(1,1) = 1

Loop Invariant

\( u \times x + v \times y = 2 \times a \times b \)

rules out candidate
Invariant Example

Least Common Multiple program:

```c
int lcm(int a, int b)
    x = a; y = b; u = b; v = a;
    while (x != y)
        if (x > y)
            x = x - y; u = b; v = v + u;
        else
            y = y - x; u = u + v;
    return (u + v) / 2;
```

lcm(1, 1) = 1
It's As If:
lcm(7, 15) = 105
lcm(7, 15) = 56

Loop Invariant

\[ u \times x + v \times y = 2 \times a \times b \]

rules out candidate
Invariants and Trust

- In our experiments, 33% of lcm candidate repairs violate the invariant
  - And each one fails a held-out benign input
- Manual inspection of the remainder reveals only trustworthy neutral edits
- In addition, by selecting those candidate repairs that respect majority invariants we simplify the implication proof
  - The repair provably maintains key invariants from the original (and possibly adds more)
Evidence and Assessments

- Approximations to the Oracle Problem
- A post-repair system is correct when ...
  - It produces similar binary *execution signals* to previous known-good runs
  - It *passes tests* adapted from similar known-good methods
  - It provably maintains non-spurious known-good *invariants*
- These can be assessed regardless of how the software repair is produced
Summary

- We desire trusted resilient systems
- Repair provides resilience but not trust
- We propose three modalities of evidence
  - Models of Execution Signals
  - Targeted Differential Testing
  - Proven Inferred Invariants
- These can provide an expanded assessment of trust in a resilient repaired system