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

Automatically Finding Patches Using Genetic Programming *

Westley Weimer
University of Virginia
weimer@virginia.edu

ThanhVu Nguyen
University of New Mexico
tnguyen@cs.unm.edu

Claire Le Goues
University of Virginia
legoues@virginia.edu

Stephanie Forrest
University of New Mexico
forrest@cs.unm.edu

Abstract

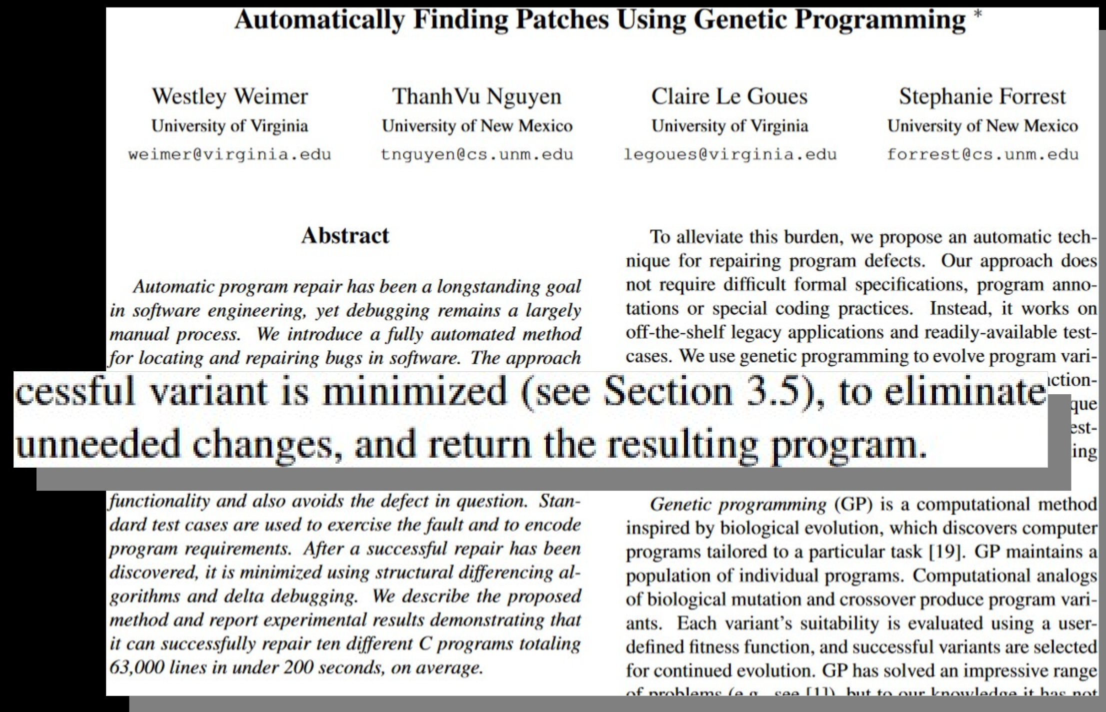
Automatic program repair has been a longstanding goal in software engineering, yet debugging remains a largely manual process. We introduce a fully automated method for locating and repairing bugs in software. The approach works on off-the-shelf legacy applications and does not require formal specifications, 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 63,000 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 testcases. 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 testcases that encode required program behavior, and a failing negative testcase 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

Program Repair Quality

- GenProg '09 – minimize
  - Remove spurious insertions



Automatically Finding Patches Using Genetic Programming *

Westley Weimer, University of Virginia, weimer@virginia.edu
ThanhVu Nguyen, University of New Mexico, tnguyen@cs.unm.edu
Claire Le Goues, University of Virginia, legoues@virginia.edu
Stephanie Forrest, University of New Mexico, forrest@cs.unm.edu

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

cessful variant is minimized (see Section 3.5), to eliminate unneeded changes, and return the resulting program.

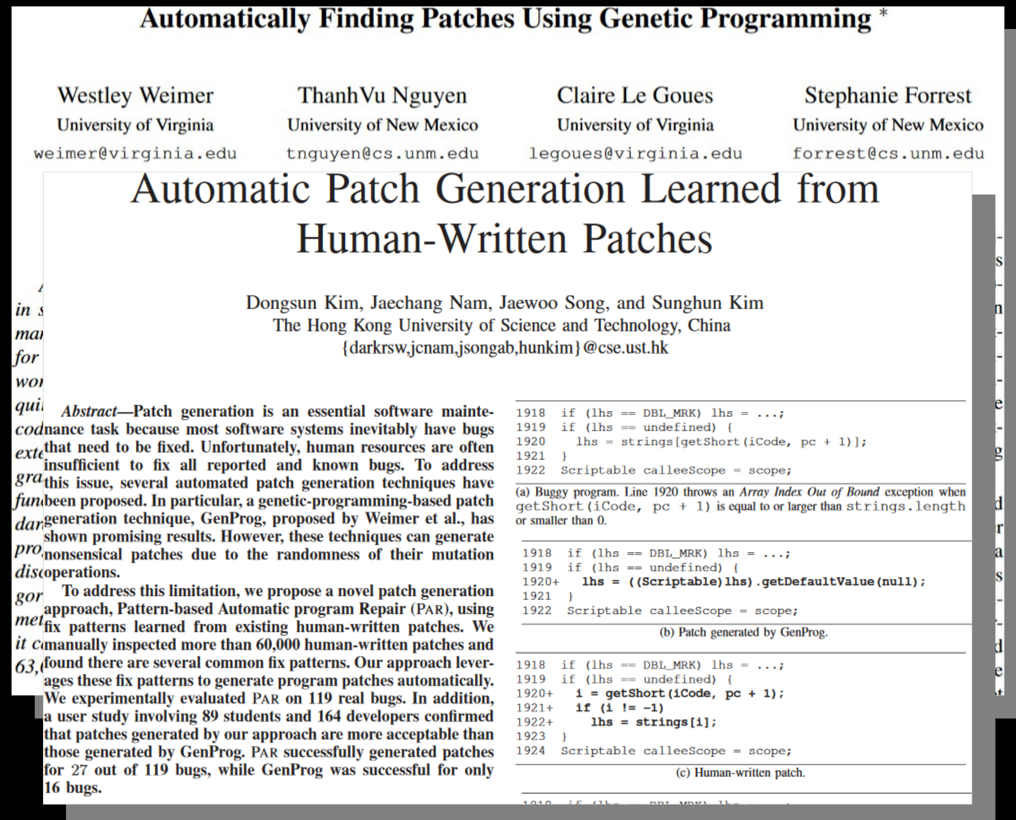
functionality and also avoids the defect in question. Standard test cases are used to exercise the fault and to encode program requirements. After a successful repair has been discovered, it is minimized using structural differencing algorithms and delta debugging. We describe the proposed method and report experimental results demonstrating that it can successfully repair ten different C programs totaling 63,000 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 testcases. We use genetic programming to evolve program vari-

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

Program Repair Quality

- GenProg '09 - minimize
- PAR '13 – human changes
    - Mutation operations based on historical human edits


Automatically Finding Patches Using Genetic Programming *

Westley Weimer, University of Virginia, weimer@virginia.edu
ThanhVu Nguyen, University of New Mexico, tnguyen@cs.unm.edu
Claire Le Goues, University of Virginia, legoues@virginia.edu
Stephanie Forrest, University of New Mexico, forrest@cs.unm.edu

Automatic Patch Generation Learned from Human-Written Patches

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim
The Hong Kong University of Science and Technology, China
{darkrsw,jcnam,jsongab,hunkim}@cse.ust.hk

Abstract—Patch generation is an essential software maintenance task because most software systems inevitably have bugs that need to be fixed. Unfortunately, human resources are often insufficient to fix all reported and known bugs. To address this issue, several automated patch generation techniques have been proposed. In particular, a genetic-programming-based patch generation technique, GenProg, proposed by Weimer et al., has shown promising results. However, these techniques can generate nonsensical patches due to the randomness of their mutation operations.

To address this limitation, we propose a novel patch generation approach, Pattern-based Automatic program Repair (PAR), using fix patterns learned from existing human-written patches. We manually inspected more than 60,000 human-written patches and found there are several common fix patterns. Our approach leverages these fix patterns to generate program patches automatically. We experimentally evaluated PAR on 119 real bugs. In addition, a user study involving 89 students and 164 developers confirmed that patches generated by our approach are more acceptable than those generated by GenProg. PAR successfully generated patches for 27 out of 119 bugs, while GenProg was successful for only 16 bugs.

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920    lhs = strings[getShort(iCode, pc + 1)];
1921  }
1922  Scriptable calleeScope = scope;
```

(a) Buggy program. Line 1920 throws an *Array Index Out of Bound* exception when getShort(iCode, pc + 1) is equal to or larger than strings.length or smaller than 0.

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+   lhs = ((Scriptable)lhs).getDefaultValue(null);
1921  }
1922  Scriptable calleeScope = scope;
```

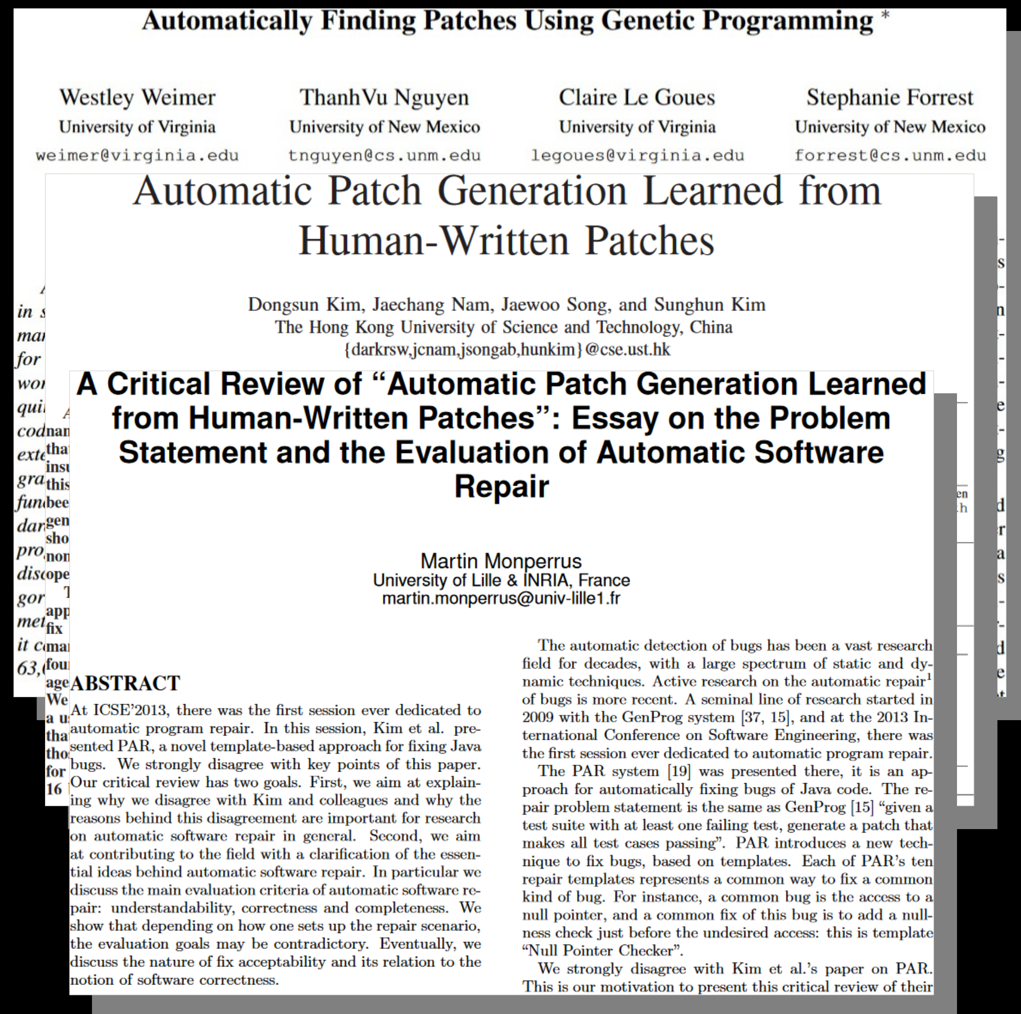
(b) Patch generated by GenProg.

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+   i = getShort(iCode, pc + 1);
1921+   if (i != -1)
1922+     lhs = strings[i];
1923  }
1924  Scriptable calleeScope = scope;
```

(c) Human-written patch.

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



Automatically Finding Patches Using Genetic Programming *

Westley Weimer, University of Virginia, weimer@virginia.edu
ThanhVu Nguyen, University of New Mexico, tnguyen@cs.unm.edu
Claire Le Goues, University of Virginia, legoues@virginia.edu
Stephanie Forrest, University of New Mexico, forrest@cs.unm.edu

Automatic Patch Generation Learned from Human-Written Patches

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim
The Hong Kong University of Science and Technology, China
{darkrsw,jcnam,jsongab,hunkim}@cse.ust.hk

A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair

Martin Monperrus
University of Lille & INRIA, France
martin.monperrus@univ-lille1.fr

ABSTRACT

At ICSE'2013, there was the first session ever dedicated to automatic program repair. In this session, Kim et al. presented PAR, a novel template-based approach for fixing Java bugs. We strongly disagree with key points of this paper. Our critical review has two goals. First, we aim at explaining why we disagree with Kim and colleagues and why the reasons behind this disagreement are important for research on automatic software repair in general. Second, we aim at contributing to the field with a clarification of the essential ideas behind automatic software repair. In particular we discuss the main evaluation criteria of automatic software repair: understandability, correctness and completeness. We show that depending on how one sets up the repair scenario, the evaluation goals may be contradictory. Eventually, we discuss the nature of fix acceptability and its relation to the notion of software correctness.

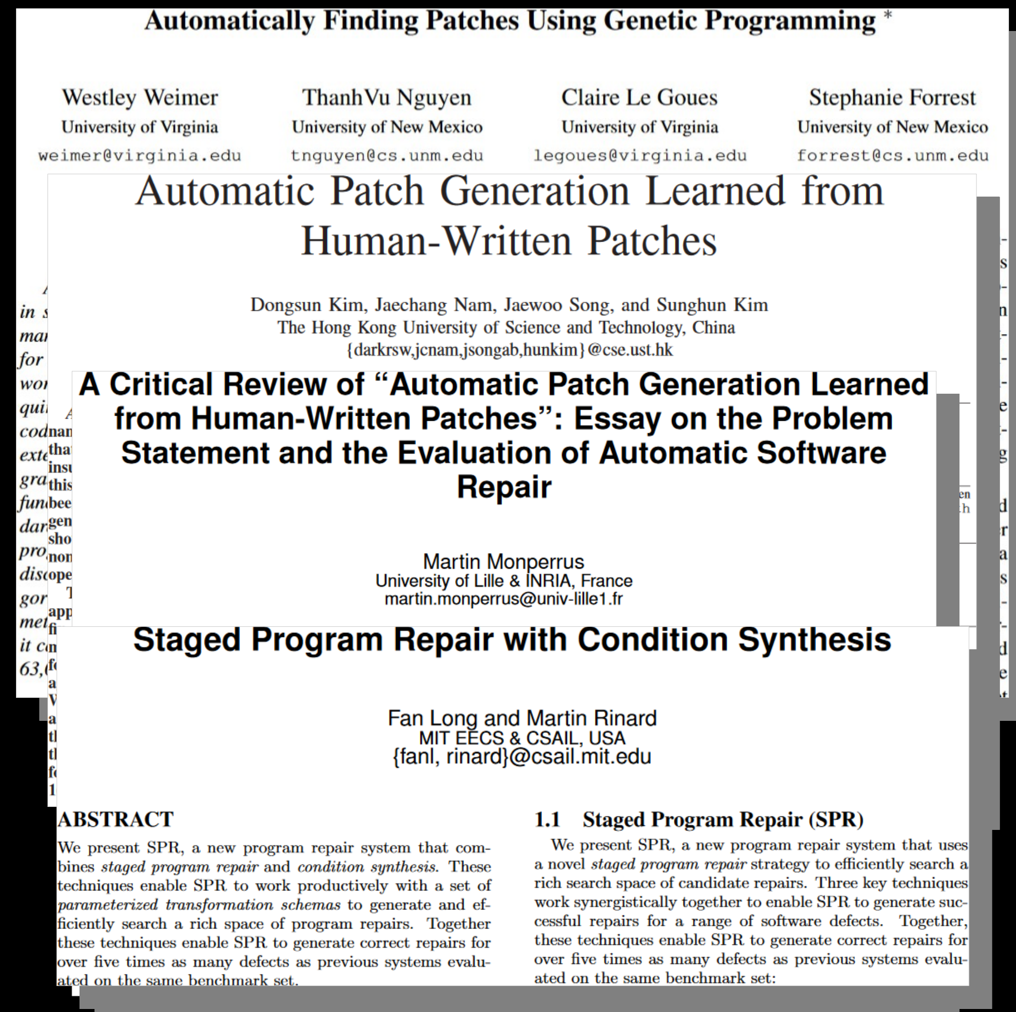
The automatic detection of bugs has been a vast research field for decades, with a large spectrum of static and dynamic techniques. Active research on the automatic repair[1] of bugs is more recent. A seminal line of research started in 2009 with the GenProg system [37, 15], and at the 2013 International Conference on Software Engineering, there was the first session ever dedicated to automatic program repair.

The PAR system [19] was presented there, it is an approach for automatically fixing bugs of Java code. The repair problem statement is the same as GenProg [15] "given a test suite with at least one failing test, generate a patch that makes all test cases passing". PAR introduces a new technique to fix bugs, based on templates. Each of PAR's ten repair templates represents a common way to fix a common kind of bug. For instance, a common bug is the access to a null pointer, and a common fix of this bug is to add a nullness check just before the undesired access: this is template "Null Pointer Checker".

We strongly disagree with Kim et al.'s paper on PAR. This is our motivation to present this critical review of their

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



Automatically Finding Patches Using Genetic Programming *

Westley Weimer, University of Virginia, weimer@virginia.edu
ThanhVu Nguyen, University of New Mexico, tnguyen@cs.unm.edu
Claire Le Goues, University of Virginia, legoues@virginia.edu
Stephanie Forrest, University of New Mexico, forrest@cs.unm.edu

Automatic Patch Generation Learned from Human-Written Patches

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim
The Hong Kong University of Science and Technology, China
{darkrsw,jcnam,jsongab,hunkim}@cse.ust.hk

A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair

Martin Monperrus
University of Lille & INRIA, France
martin.monperrus@univ-lille1.fr

Staged Program Repair with Condition Synthesis

Fan Long and Martin Rinard
MIT EECS & CSAIL, USA
{fanl, rinard}@csail.mit.edu

ABSTRACT

We present SPR, a new program repair system that combines *staged program repair* and *condition synthesis*. These techniques enable SPR to work productively with a set of *parameterized transformation schemas* to generate and efficiently search a rich space of program repairs. Together these techniques enable SPR to generate correct repairs for over five times as many defects as previous systems evaluated on the same benchmark set.

1.1 Staged Program Repair (SPR)

We present SPR, a new program repair system that uses a novel *staged program repair* strategy to efficiently search a rich search space of candidate repairs. Three key techniques work synergistically together to enable SPR to generate successful repairs for a range of software defects. Together, these techniques enable SPR to generate correct repairs for over five times as many defects as previous systems evaluated on the same benchmark set:

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

Automatically Finding Patches Using Genetic Programming *

Westley Weimer, University of Virginia, weimer@virginia.edu
ThanhVu Nguyen, University of New Mexico, tnguyen@cs.unm.edu
Claire Le Goues, University of Virginia, legoues@virginia.edu
Stephanie Forrest, University of New Mexico, forrest@cs.unm.edu

Automatic Patch Generation Learned from Human-Written Patches

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim
The Hong Kong University of Science and Technology, China
{darkrsw,jcnam,jsongab,hunkim}@cse.ust.hk

A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair

Martin Monperrus
University of Lille & INRIA, France
martin.monperrus@univ-lille1.fr

Staged Program Repair with Condition Synthesis

Fan Long and Martin Rinard
MIT EECS & CSAIL, USA
{fanl, rinard}@csail.mit.edu

Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis

Jooyong Yi   Abhik Roychoudhury
...mputing, National University of Singapore, Singapore
...taev,jooyong,abhik}@comp.nus.edu.sg

terest. 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" re-

...pairs. Unlike generate-and-validate systems such as GenProg and SPR, semantic analysis based repair techniques *synthesize* a repair based on semantic information of the

tools, such as GenProg [14], PAR [21], *relifix* [39], SemFix [26], Nopol [8], DirectFix [24] and SPR [23], to name only a few, have been introduced recently. These automated repair methods can be classified into the following two broad methodologies, i.e., search-based methodology (e.g., GenProg, PAR, and SPR) and semantics-based methodology (e.g., SemFix, Nopol, and DirectFix). Search-based repair methodology (also known as generate-and-validate methodology) searches within a search space to *generate* a repair candidate and *validate* this repair candidate against the provided test-suite. Meanwhile, the semantics-based re-

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

```
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);
}
```

“top”

“bot”

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

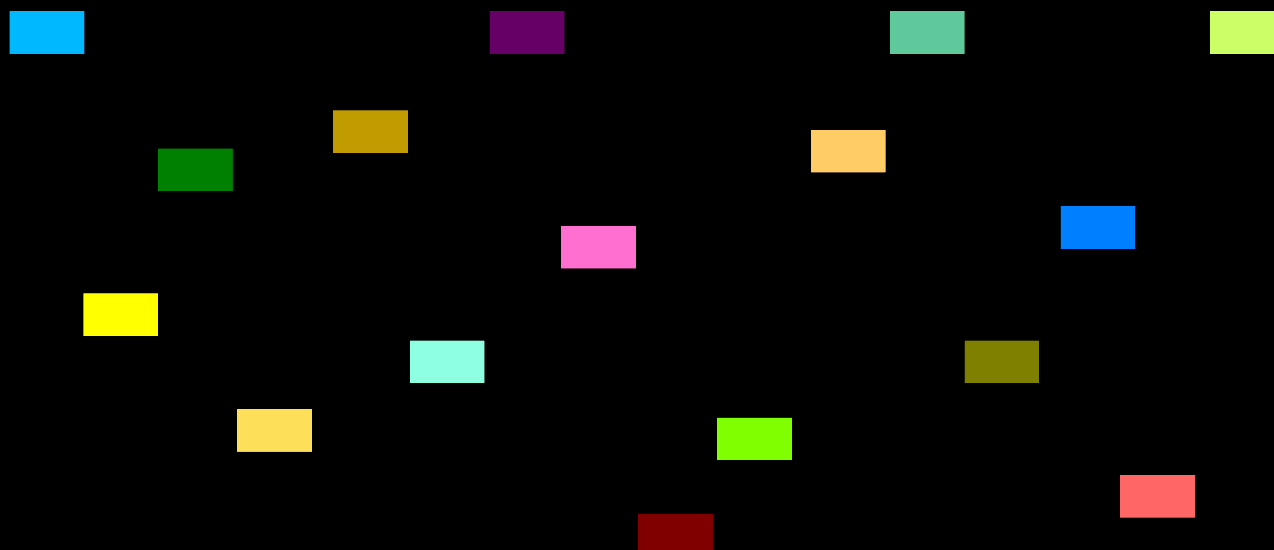
- Given one candidate repair …
- Generate a large number of neutral (or test-equivalent) alternate candidate repairs
  - Via a special directed neutral walk
- Dynamically infer and statically verify invariants of those candidate repairs
- Select repairs that respect majority invariants

Generating Alternate Repairs

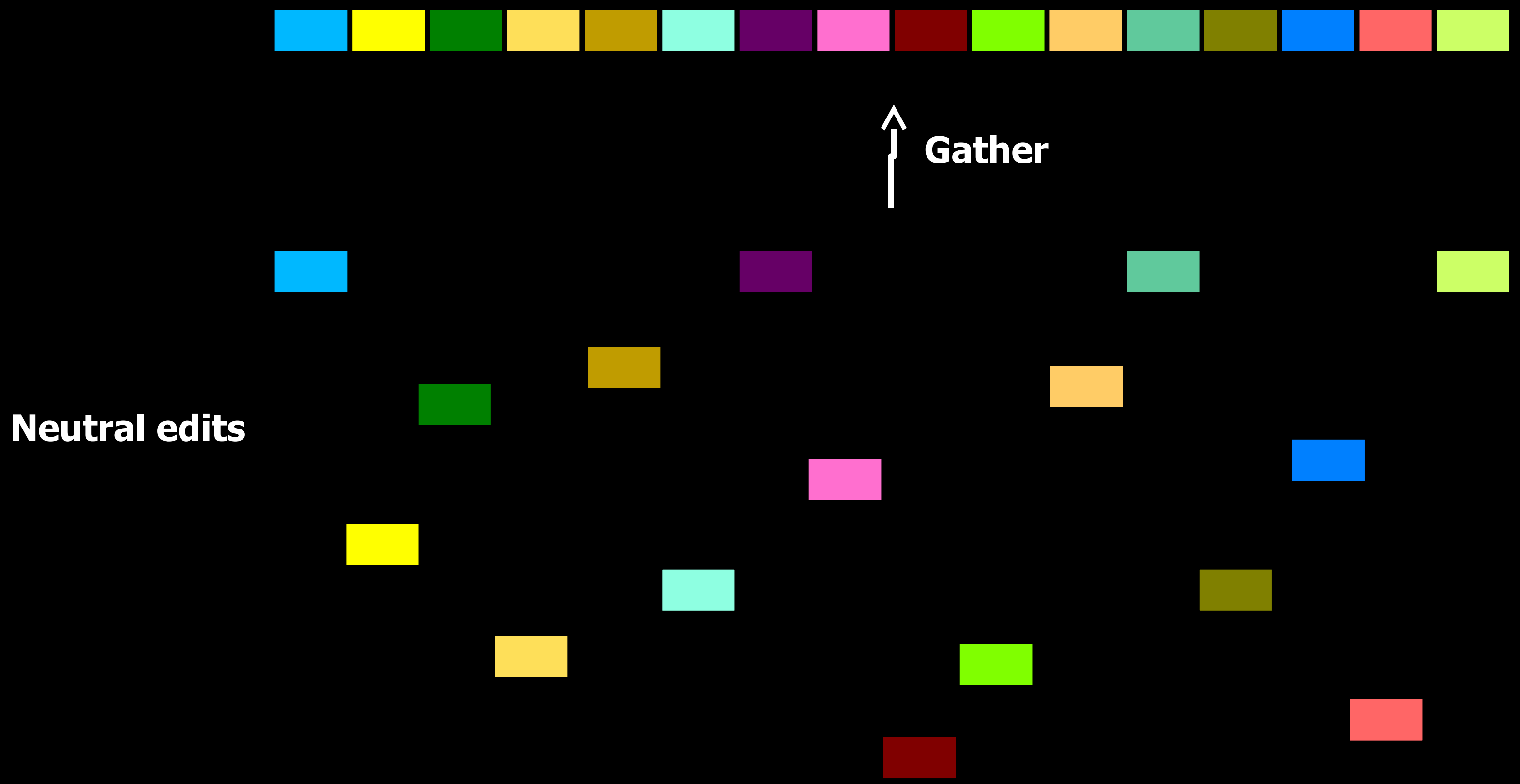
- We can generate many neutral edits
  - Changes to a program that retain behavioral equivalence with respect to a test suite
  - But may behave differently for future attacks or unconsidered benign inputs
- Cheaply generate singleton neutral edits
- Then combine (or “cluster') many of them to make a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

Directed Neutral Walk

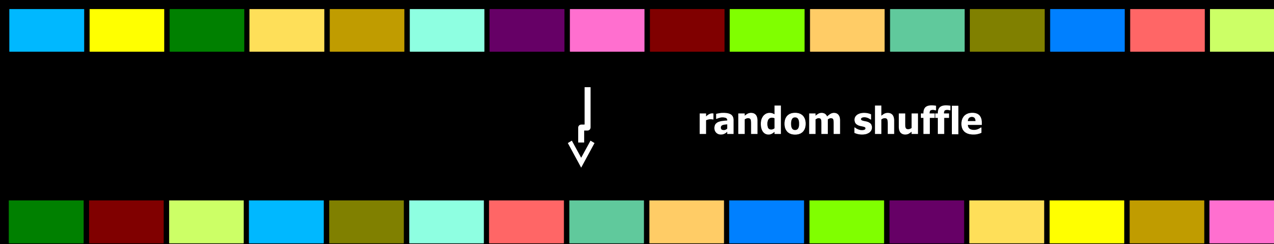
Neutral edits



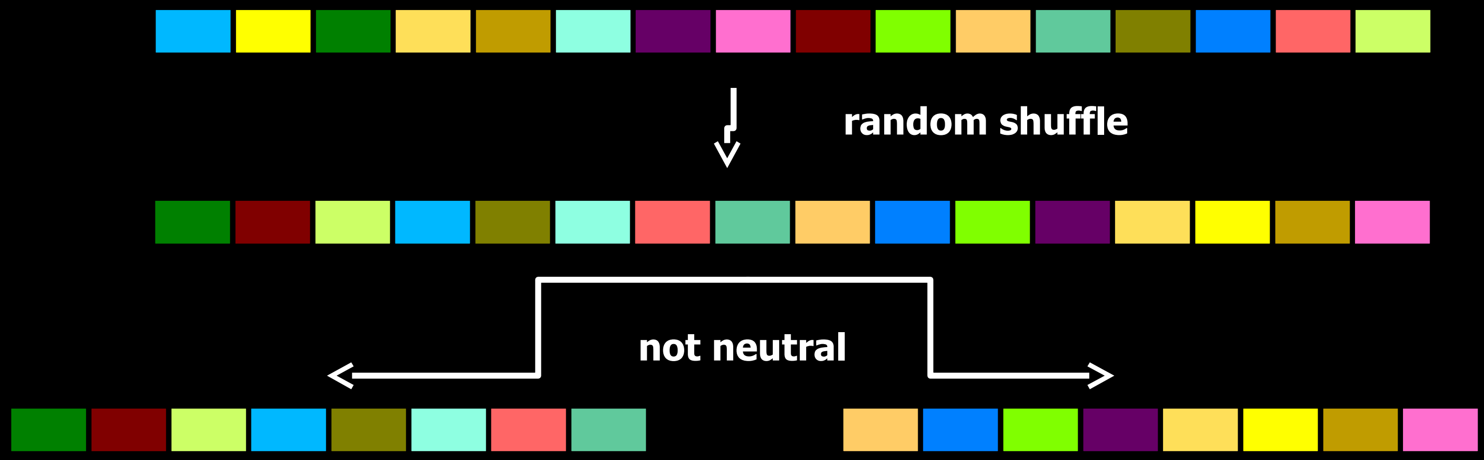
Directed Neutral Walk



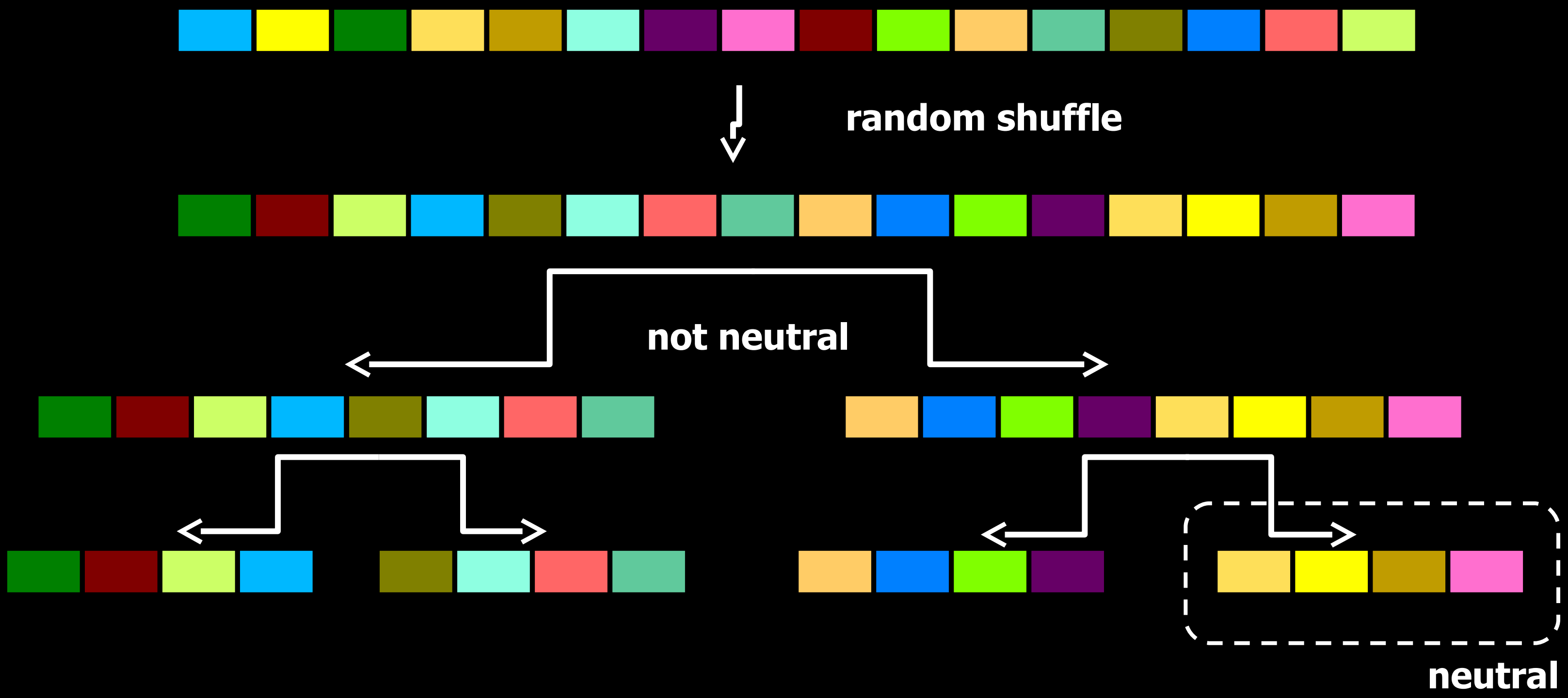
Directed Neutral Walk



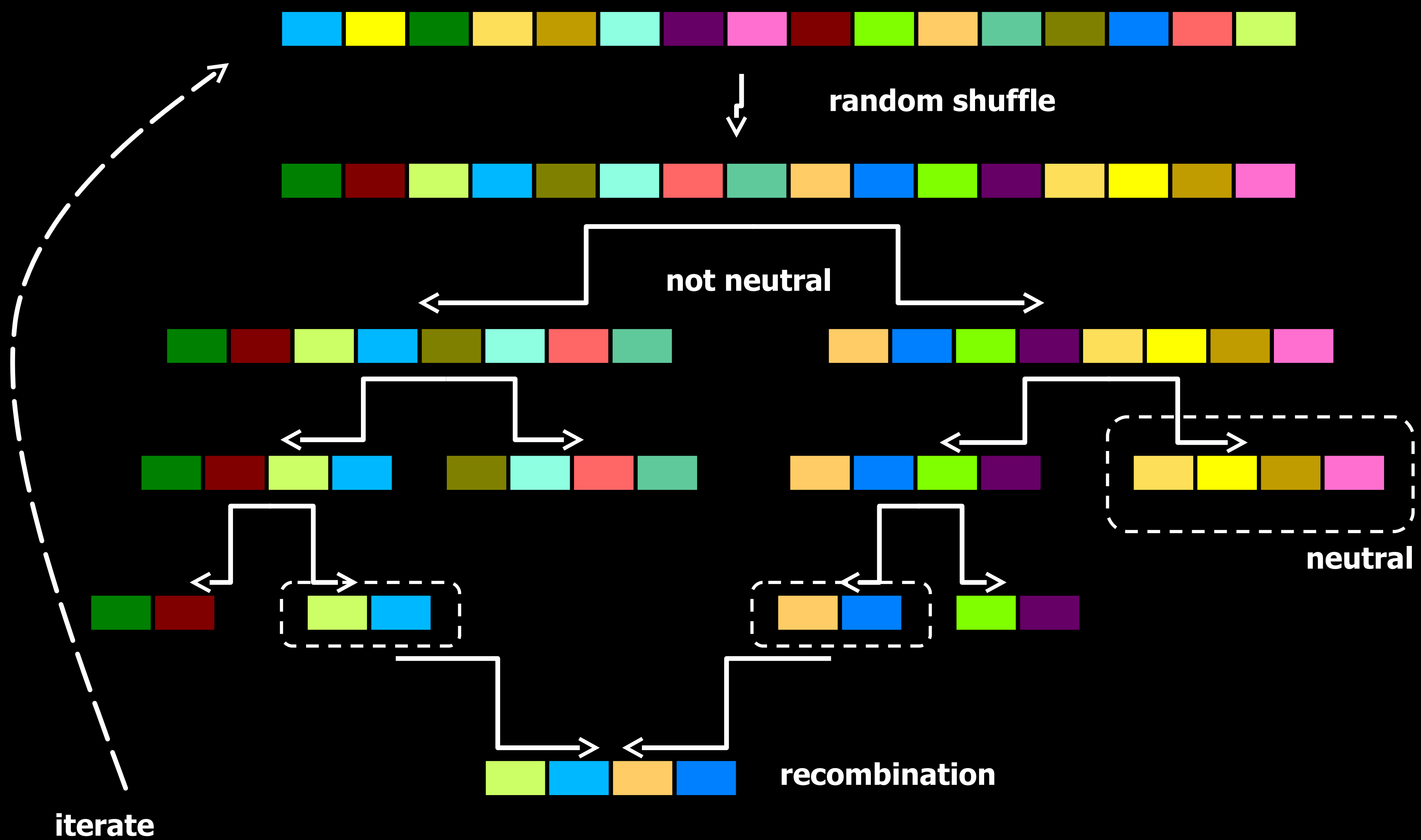
Directed Neutral Walk



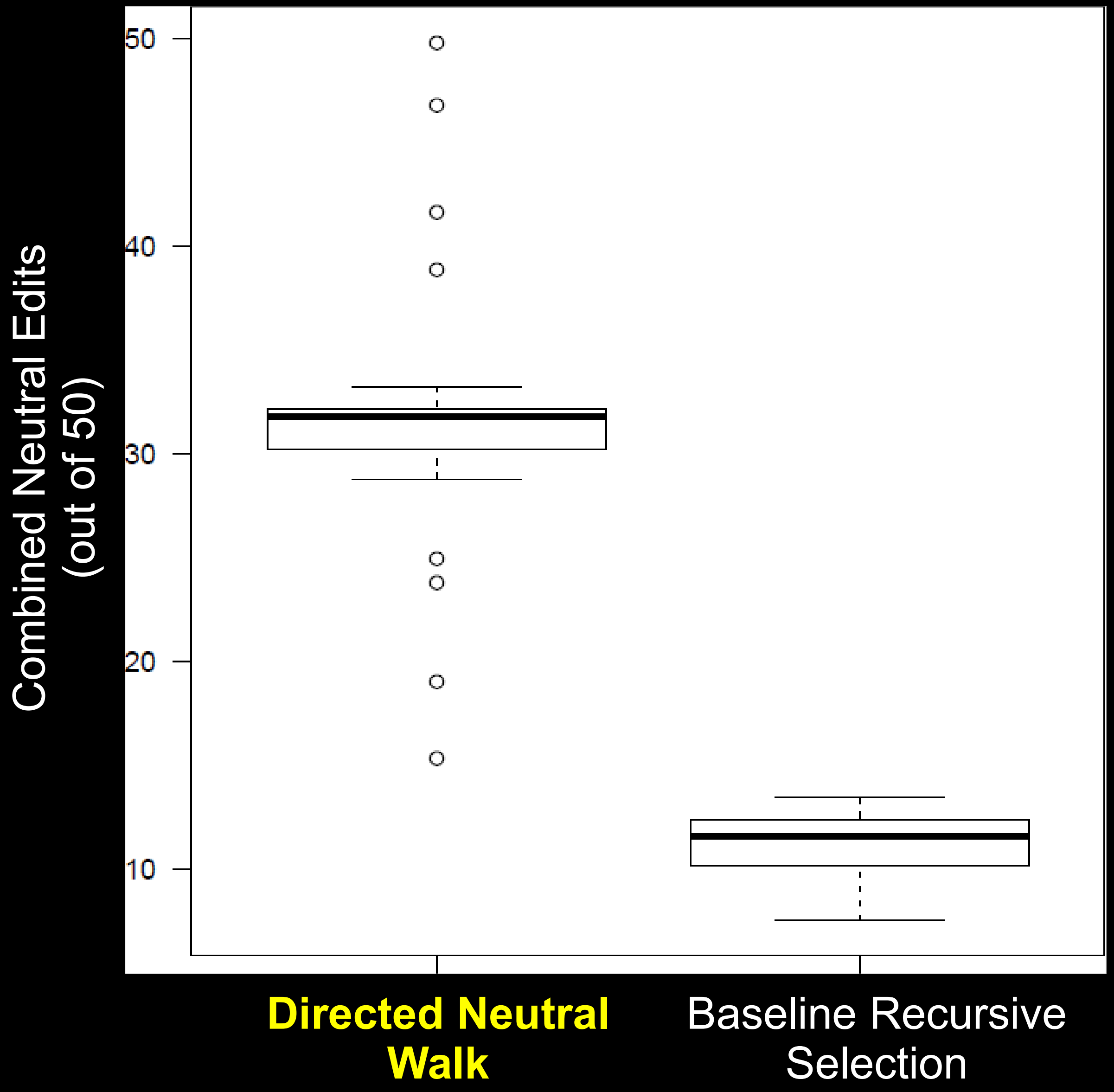
Directed Neutral Walk



Directed Neutral Walk



Effective Combination



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)

Invariant Example

Least Common Multiple program:

```
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

Least Common Multiple program:

```
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:

```
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:

```
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*x + v*y == 2*a*b

Invariant Example

Least Common Multiple program:

```
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*x + v*y == 2*a*b
rules out candidate

Invariant Example

Least Common Multiple program:

```
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*x + v*y == 2*a*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