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.

1 Introduction

Fixing bugs is a difficult, time-consuming, and manual process. Some reports place software maintenance, traditionally defined as any modification made on a system after its delivery, at 90% of the total cost of a typical software project [27]. Modifying existing code, repairing defects, and otherwise evolving software are major parts of those costs [24]. The number of outstanding software defects typically exceeds the resources available to address them [4]. Mature software projects are forced to ship with both known and unknown bugs [21] because they lack the development resources to deal with every defect. For example, in 2005, one Mozilla developer claimed that, "everyday, almost 300 bugs appear [...] far too much for only the Mozilla programmers to handle" [5, p. 363].

*This research was supported in part by National Science Foundation Grants CNS 0627523 and CNS 0716478, Air Force Office of Scientific Research grant FA9550-07-1-0532, as well as gifts from Microsoft Research. No official endorsement should be inferred.

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 of problems (e.g., see [1]), but to our knowledge it has not been used to evolve off-the-shelf legacy software.

A significant impediment for an evolutionary algorithm like GP is the potentially infinite-size search space it must sample to find a correct program. To address this problem, we introduce two key innovations. First, we restrict the algorithm to only produce changes that are based on structures in other parts of the program. In essence, we hypothesize that a program that is missing important functionality (e.g., a null check) will be able to copy and adapt it from another location in the program. Second, we constrain the genetic operations of mutation and crossover to operate only on the region of the program that is relevant to the error (that is, the portions of the program that were on the execution path that produced the error). Combining these insights, we demonstrate automatically generated repairs for ten C programs totaling 63,000 lines of code.

We use GP to maintain a population of variants of that program. Each variant is represented as an abstract syntax tree (AST) paired with a weighted program path. We modify variants using two genetic algorithm operations, crossover and mutation, specifically targeted to this repre-

sentation; each modification produces a new abstract syntax tree and weighted program path. The fitness of each variant is evaluated by compiling the abstract syntax tree and running it on the testcases. Its final fitness is a weighted sum of the positive and negative testcases it passes. We stop when we have evolved a program variant that passes all of the testcases. Because GP often introduces irrelevant changes or dead code, we use tree-structured difference algorithms and delta debugging techniques in a post-processing step to generate a 1-minimal patch that, when applied to the original program, causes it to pass all of the testcases.

The main contributions of this paper are:

- Algorithms to find and minimize program repairs based on testcases that describe desired functionality.
- A novel and efficient representation and set of operations for scaling GP in this domain. To the best of our knowledge, this is the first use of GP to scale to and repair real unannotated programs.
- Experimental results showing that the approach can generate repairs for different kinds of defects in ten programs from multiple domains.

The structure of the paper is as follows. In Section 2 we give an example of a simple program and how it might be repaired. Section 3 describes the technique in detail, including our program representation (Section 3.2), genetic operators (Section 3.3), fitness function (Section 3.4) and approach to repair minimization (Section 3.5). We empirically evaluate our approach in Section 4, including discussions of success rate (Section 4.4) and repair quality (Section 4.3). We discuss related work in Section 5 and conclude.

2 Motivating Example

This section uses a small example program to illustrate the important design decisions in our approach. Consider the C program below, which implements Euclid's greatest common divisor algorithm:

```
1  /* requires: a >= 0, b >= 0 */
2  void gcd(int a, int b) {
3    if (a == 0) {
4      printf("%d", b);
5    }
6    while (b != 0)
7      if (a > b)
8        a = a - b;
9      else
10       b = b - a;
11   printf("%d", a);
12   exit(0);
13 }
```

The program has a bug; when `a` is zero and `b` is positive, the program prints out the correct answer but then loops forever on lines 6–9–10. The code could have other bugs: it does not handle negative inputs gracefully. In order to repair the program we must understand what it is supposed to be doing; we use testcases to codify these requirements. For example, we might use the testcase `gcd(0,55)` with desired terminating output `55`. The program above fails this testcase, which helps us identify the defect to be repaired.

Our algorithm attempts to automatically repair the defect by searching for valid variants of the original program. Searching randomly through possible program modifications for a repair may not yield a desirable result. Consider the following program variant:

```
1  void gcd_2(int a, int b) {
2    printf("%d", b);
3    exit(0);
4  }
```

This `gcd_2` variant passes the `gcd(0,55)` testcase, but fails to implement other important functionality. For example, `gcd_2(1071,1029)` produces `1029` instead of `21`. Thus, the variants must pass the negative testcase while retaining other core functionality. This is enforced through positive testcases, such as terminating with output `21` on input `gcd(1071,1029)`. In general, several positive testcases will be necessary to encode the requirements, although in this simple example a single positive testcase suffices.

For large programs, we would like to bias the modifications towards regions of code that are likely to change behavior on the negative testcase without damaging performance on the positive testcases. We thus instrument the program to record all lines visited when processing the testcases. The positive testcase `gcd(1071,1029)` visits lines 2–3 and 6–13. The negative testcase `gcd(0,55)` visits lines 2–5, 6–7, and 9–10. When selecting portions of the program to modify, we favor those that were visited during the negative testcase and were not also visited during the positive one. In this example, repairs are focused on lines 4–5.

Even if we know where to change the program, the number of possible changes is still huge, and this has been a significant impediment for GP in the past. We could add arbitrary code, delete existing code, or change existing code into new arbitrary code. We make the assumption that most defects can be repaired by adopting existing code from another location in the program. In practice, a program that makes a mistake in one location often handles the situation correctly in another [13]. As a simple example, a program missing a null check or an array bounds check is likely to have a similar working check somewhere else that can be used as a template. When mutating a program we may insert, delete or modify statements, but we insert only code that is similar in structure to existing code. Thus, we will not insert an arbitrary `if` conditional, but we might insert `if(a==0)` or `if(a>b)` because they already appear in the program. Similarly, we might insert `printf("%d",a)`, `a=a-b`, `b=b-a`, `printf("%d",b)`, or `exit(0)`, but not arbitrary statements.

Given the bias towards modifying lines 4–5 and our preference for insertions similar to existing code, it is reasonable to consider inserting `exit(0)` and `a=a-b` between lines 4 and 5, which yields:

```
1  void gcd_3(int a, int b) {
2    if (a == 0) {
3      printf("%d", b);
4      exit(0);              // inserted
5      a = a - b;            // inserted
6    }
7    while (b != 0)
8      if (a > b)
9        a = a - b;
10     else
11       b = b - a;
12   printf("%d", a);
13   exit(0);
14 }
```

This `gcd_3` variant passes all of the positive testcases and also passes the negative testcase; we call it the primary repair. We could return it as the final repair. However, the `a = a - b` inserted on line 5 is extraneous. The GP method often produces such spurious changes, which we minimize away in a final postprocessing step. We consider all of the changes between the original `gcd` and the primary repaired variant `gcd_3`, retain the minimal subset of changes that, when applied to `gcd`, allow it to pass all positive and negative testcases. This minimal patch is the final repair:

```
3      if (a == 0) {
4        printf("%d", b);
5  +     exit(0);
6      }
7      while (b != 0) {
```

In the next section, we generalize this procedure.

3 Genetic Programming for Software Repair

The core of our method is a GP that repairs programs by selectively searching through the space of nearby program variants until it discovers one that avoids known defects and retains key functionality. We use a novel GP representation and make assumptions about the probable nature and location of the necessary repair to make the search more efficient. Given a defective program, we address five issues:

1. **What is it doing wrong?** We take as input a set of *negative testcases* that characterizes a fault. The input program fails all negative testcases.
2. **What is it supposed to do?** We take as input a set of *positive testcases* that encode functionality requirements. The input program passes all positive testcases.
3. **Where should we change it?** We favor changing program locations visited when executing the negative testcases and avoid changing program locations visited when executing the positive testcases.

Input: Program P to be repaired.
Input: Set of positive testcases $PosT$.
Input: Set of negative testcases $NegT$.
Output: Repaired program variant.

1: $Path_{PosT} \leftarrow \bigcup_{p \in PosT}$ statements visited by $P(p)$
2: $Path_{NegT} \leftarrow \bigcup_{n \in NegT}$ statements visited by $P(n)$
3: $Path \leftarrow \mathsf{set\_weights}(Path_{NegT}, Path_{PosT})$
4: $Popul \leftarrow \mathsf{initial\_population}(P, \mathsf{pop\_size})$
5: **repeat**
6: $\quad Viable \leftarrow \{\langle P, Path_P, f\rangle \in Popul \mid f > 0\}$
7: $\quad Popul \leftarrow \emptyset$
8: $\quad NewPop \leftarrow \emptyset$
9: $\quad$ **for all** $\langle p_1, p_2\rangle \in \mathsf{sample}(Viable, \mathsf{pop\_size}/2)$ **do**
10: $\quad\quad \langle c_1, c_2\rangle \leftarrow \mathsf{crossover}(p_1, p_2)$
11: $\quad\quad NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$
12: $\quad$ **end for**
13: $\quad$ **for all** $\langle V, Path_V, f_V\rangle \in NewPop$ **do**
14: $\quad\quad Popul \leftarrow Popul \cup \{\mathsf{mutate}(V, Path_V)\}$
15: $\quad$ **end for**
16: **until** $\exists \langle V, Path_V, f_V\rangle \in Popul \;.\; f_V = \mathsf{max\_fitness}$
17: **return** $\mathsf{minimize}(V, P, PosT, NegT)$

Figure 1. High-level pseudocode for our technique. Lines 5–16 describe the GP search for a feasible variant. Subroutines such as $\mathsf{mutate}(V, Path_V)$ are described subsequently.

4. **How should we change it?** We insert, delete, and swap program statements and control flow. We favor insertions based on the existing program structure.
5. **When are we finished?** The primary repair is the first variant that passes all positive and negative testcases. We minimize the differences between it and the original input program to produce the final repair.

Pseudocode for the algorithm is given in Figure 1. Lines 1–2 determine the paths visited by the program on the input testcases. Line 3 combines the weights from those paths (see Section 3.2). With these preprocessing steps complete, Line 4 constructs an initial GP population based on the input program. Lines 5–16 encode the main GP loop (see Section 3.1), which searches for a feasible variant. On each iteration, we remove all variants that fail every testcase (line 6). We then take a weighted random sample of the remaining variants (line 9), favoring those variants that pass more of the testcases (see Section 3.4). We apply the crossover operator (line 10; see Section 3.3) to the selected variants; each pair of parent variants produces two child variants. We include the parent and child variants in the population and then apply the mutation operator to each variant (line 14; see Section 3.3); this produces the population for the next generation. The algorithm terminates when it produces a variant that passes all of the testcases (line 16). The suc-

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

3.1 Genetic Programming (GP)

As mentioned earlier, GP is a stochastic search method based on principles of biological evolution [14, 19]. GP operates on and maintains a population comprised of different programs, referred to as *individuals* or *chromosomes*. In GP, each chromosome is a tree-based representation of a program. The *fitness*, or desirability, of each chromosome, is evaluated via an external *fitness function*—in our application fitness is assessed via the test cases. Once fitness is computed, high-fitness individuals are *s*elected to be copied into the next generation. Variations are introduced through computational analogies to the biological processes of mutation and crossover (see below). These operations create a new generation and the cycle repeats. Details of our GP implementation are given in the following subsections.

3.2 Program Representation

We represent each individual (candidate program) as a pair containing:

1. An *abstract syntax tree (AST)* including all of the statements in the program.

2. A *weighted path* through that program. The weighted path is a list of pairs, each pair containing a statement in the program and a weight based on that statement's occurrences in various testcases.

The specification of what constitutes a statement is critical because the GP operators are defined over statements. Our implementation uses the CIL toolkit for manipulating C programs, which reduces multiple statement and expression types in C to a small number of high-level abstract syntax variants. In CIL's terminology, Instr, Return, If and Loop are defined as statements [23]; this includes all assignments, function calls, conditionals, and looping constructs. An example of C syntax not included in this definition is `goto`. The genetic operations will thus never delete, insert or swap a lone `goto` directly; however, the operators might insert, delete or swap an entire loop or conditional block that contains a `goto`. With this simplified form of statement, we use an off-the-shelf CIL AST. To find the statements visited along a program execution (lines 1–2 of Figure 1), we apply a program transformation, assigning each statement element a unique number and inserting a `fprintf` call that logs the visit to that statement.

The *weighted path* is a set of $\langle$ statement, weight $\rangle$ pairs that guide the GP search. We assume that a statement visited at least once during a negative testcase is a reasonable candidate for repair. We do not assume that a statement visited frequently (e.g., because it is in a loop) is more likely to be a good repair site. We thus remove all duplicates from each list of statements. However, we retain the visit order in our representation (see Crossover description below). If there were no positive testcases, each statement visited along a negative testcase would be a reasonable repair candidate, so the initial weight on every statement would be 1.0. We modify these weights using the positive testcases. set_weights$(Path_{NegT}, Path_{PosT})$ in Figure 1 sets the weight of every statement on the path that is also visited in at least one positive testcase equal to a parameter W_{Path}. Taking $W_{Path} = 0$ prevents us from considering any statement visited on a positive testcase; values such as $W_{Path} = 0.01$ typically work better (see Section 4.4).

Each GP-generated program has the same number of pairs and the same sequence of weights in its weighted path as the original program. By adding the weighted path to the AST representation, we can constrain the evolutionary search to a small subset of the complete program tree by focusing the genetic operators on relevant code locations. In addition, the genetic operators are not allowed to invent completely new statements. Instead, they "borrow" statements from the rest of the program tree. These modifications allow us to address the longstanding scaling issues in GP [16] and apply the method to larger programs than were previously possible.

3.3 Selection and Genetic Operators

Selection. There are many possible selection algorithms in which more fit individuals are allocated more copies in the next generations than less fit ones. For our initial prototype we used *stochastic universal samplying* (SUS) [12], in which each individual's probability of selection is directly proportional to its relative fitness in the population. We use SUS to select pop_size$/2$ new members of the population. The code from lines 6–9 of Figure 1 implements the selection process. We discard individuals with fitness 0 (i.e., variants that do not compile or variants that pass no test cases), placing the remainder in $Viable$ on line 6. These individuals form the mating pool and are used as parents in the crossover operation (below).

We use two GP operators, mutation and crossover, to create new program variants from this mating pool. Mutation has a small chance of changing any particular statement along the weighted path. Crossover combines the "first part" of one variant with the "second part" of another, where "first" and "second" are relative to the weighted path.

Mutation. Figure 2 shows the high-level pseudocode for our *mutation* operator. Mutation is constrained to the statements on the weighted path (line 1). Each location on the weighted path is considered for mutation with probability

Input: Program P to be mutated.
Input: Path $Path_P$ of interest.
Output: Mutated program variant.

```
1:  for all ⟨stmt_i, prob_i⟩ ∈ Path_P do
2:    if rand(0,1) ≤ prob_i ∧ rand(0,1) ≤ W_mut then
3:      let op = choose({insert, swap, delete})
4:      if op = swap then
5:        let stmt_j = choose(P)
6:        Path_P[i] ← ⟨stmt_j, prob_i⟩
7:      else if op = insert then
8:        let stmt_j = choose(P)
9:        Path_P[i] ← ⟨{stmt_i; stmt_j}, prob_i⟩
10:     else if op = delete then
11:       Path_P[i] ← ⟨{}, prob_i⟩
12:     end if
13:   end if
14: end for
15: return ⟨P, Path_P, fitness(P)⟩
```

Figure 2. Our mutation operator. Updates to $Path_P$ also update the AST P.

equal to its path weight. A statement occurring on negative testcases but not on positive testcases ($prob_i = 1$) is always considered for mutation, whereas a statement that occurs on both positive and negative testcases is less likely to be mutated ($prob_i = W_{Path}$). Even if a statement is considered for mutation, only a few mutations actually occur, as determined by the global mutation rate (a parameter called W_{mut}) Ordinarily, mutation operations involve single bit flips or simple symbolic substitutions. Because our primitive unit is the statement, our mutation operator is more complicated, consisting of either a deletion (the entire statement is deleted), an insertion (another statement is inserted after it), or a swap with another statement. In the current implementation we choose from these three options with uniform random probability (1/3, 1/3, 1/3).

In the case of a swap, a second statement $stmt_j$ is chosen uniformly at random from anywhere in the program — not just from along the path. This reflects our intuition about related changes; a program missing a null check may not have one along the weighted path, but probably has one somewhere else in the program. The ith element of $Path_P$ is replaced with a pair consisting of $stmt_j$ and the original weight $prob_i$; the original weight is retained because $stmt_j$ can be off the path and thus have no weight of its own. Changes to statements in $Path_P$ are reflected in its corresponding AST P. We handle insertions by transforming $stmt_i$ into a block statement that contains $stmt_i$ followed by $stmt_j$. Deletions are handled similarly by transforming $stmt_i$ into an empty block statement. We replace with nothing rather than deleting to maintain the invariant of uniform path lengths across all variants. Note that a "deleted" statement may be selected as $stmt_j$ in a later mutation operation.

Input: Parent programs P and Q.
Input: Paths $Path_P$ and $Path_Q$.
Output: Two new child program variants C and D.

```
1:  cutoff ← choose(|Path_P|)
2:  C, Path_C ← copy(P, Path_P)
3:  D, Path_D ← copy(Q, Path_Q)
4:  for i = 1 to |Path_P| do
5:    if i > cutoff then
6:      let ⟨stmt_p, prob⟩ = Path_P[i]
7:      let ⟨stmt_q, prob⟩ = Path_Q[i]
8:      if rand(0,1) ≤ prob then
9:        Path_C[i] ← Path_Q[i]
10:       Path_D[i] ← Path_P[i]
11:     end if
12:   end if
13: end for
14: return ⟨C, Path_C, fitness(C)⟩,⟨D, Path_D, fitness(D)⟩
```

Figure 3. Our crossover operator. Updates to $Path_C$ and $Path_D$ update the ASTs C and D.

Crossover. Figure 3 shows the high-level pseudocode for the *crossover* operator. Only statements along the weighted paths are crossed over. We choose a cutoff point along the paths (line 1) and swap all statements after the cutoff point. For example, on input $[P_1, P_2, P_3, P_4]$ and $[Q_1, Q_2, Q_3, Q_4]$ with cutoff 2, the child variants are $C = [P_1, P_2, Q_3, Q_4]$ and $D = [Q_1, Q_2, P_3, P_4]$. Each child thus combines information from both parents. In any given generation, a variant will be the parent in at most one crossover operation. A unique aspect of our crossover operator is that crossover always takes place between an individual from the current population and the original parent program, sometimes referred to as *crossing back*.

A second unusual feature of our crossover operator is that instead of crossing over all the statements before the cutoff point, it performs a crossover swap for a given statement with probability given by the path weight for that statement. Note that on lines 6–8, the weight from $Path_P[i]$ must be the same as the weight from $Path_Q[i]$ because of our representation invariant.

3.4 Fitness Function

Given an input program, the *fitness function* returns a number indicating the acceptability of the program. The fitness function is used by the selection algorithm to determine which variants survive to the next iteration (generation), and it is used as a termination criterion for the search. Our fitness function computes a weighted sum of all testcases passed by a variant. We first compile the variant's AST to an executable program, and then record which test-

cases are passed by that executable:

$$\begin{aligned}\mathsf{fitness}(P) &= W_{PosT} \times |\{t \in PosT \mid P \text{ passes } t\}| \\ &+ W_{NegT} \times |\{t \in NegT \mid P \text{ passes } t\}|\end{aligned}$$

Each successful positive test is weighted by the global parameter W_{PosT}; each successful negative test is weighted by the global parameter W_{NegT}. A program variant that does not compile receives a fitness of zero; 32.19% of variants failed to compile in our experiments. The search terminates when a variant maximizes the fitness function by passing all testcases. The weights W_{PosT} and W_{NegT} should be positive values; we discuss particular choices in Section 4.4.

For full safety, the testcase evaluations should be run in a virtual machine, `chroot(2)` jail, or similar sandbox. Standard software fault isolation through address spaces may be insufficient. For example, a program that removes a temporary file may in theory evolve into a variant that deletes every file in the filesystem. In addition, the standard regression testing practice of limiting execution time to prevent run-away processes is even more important in this setting, where program variants may contain infinite loops.

The fitness function encodes software requirements at the testcase level. The negative testcases encode the fault to be repaired and the positive testcases encode necessary functionality that cannot be sacrificed. Note that including too many positive testcases could make the fitness evaluation process inefficient and constrain the search space, while including too few may lead to a repair that sacrifices important functionality; Section 4.3 investigates this topic.

Because the fitness of each individual in our GP is independent of other individuals, fitness calculations can be parallelized. Fitness evaluation for a single variant is similarly parallel with respect to the number of testcases. In our prototype implementation, we were able to take advantage of multicore hardware with trivial fork-join directives in the testcase shell scripts (see Section 4).

3.5 Repair Minimization

Once a variant is discovered that passes all of the testcases we minimize the repair before presenting it to developers. Due to the randomness in the mutation and crossover algorithms, it is likely that the successful variant will include irrelevant changes that are difficult to inspect for correctness. We thus wish to produce a *patch*, a list of edits that, when applied to the original program, repair the defect without sacrificing required functionality. Previous work has shown that defects associated with such a patch are more likely to be addressed [31]. We combine insights from delta debugging and tree-structured distance metrics to minimize the repair. Intuitively, we generate a large patch by taking the difference between the variant and the original, and then discard every part of that patch we can while still passing all test cases.

We cannot use standard `diff` because its line-level patches encode program concrete syntax, rather than program abstract syntax, and are thus inefficient to minimize. For example, our variants often include changes to control flow (e.g., `if` or `while` statements) for which both the opening brace `{` and the closing brace `}` must be present; throwing away part of such a patch results in a program that does not compile. We choose not to record the genetic programming operations performed to obtain the variant as an edit script because such operations often overlap and the resulting script is quite long. Instead, we use a version of the DIFFX XML difference algorithm [2] modified to work on CIL ASTs. This generates a list of tree-structured edit operations between the variant and the original. Unlike a standard line-level patch, tree-level edits include operations such as "move the subtree rooted at node X to become the Yth child of node Z". Thus, failing to apply part of a tree-structured patch will never result in an ill-formed program at the concrete syntax level (although it may still result in an ill-formed program at the semantic type-checking level).

Once we have a set of edits that can be applied together or separately to the original program we minimize that list. Considering all subsets of the set of edits is infeasible; for example, the edit script for the first repair generated by our algorithm on the "ultrix look" program in Section 4 is 32 items long (the `diff` is 52 lines long). Instead, we use delta debugging [32]. Delta debugging finds a small "interesting" subset of a given set for an external notion of interesting, and is typically used to minimize compiler testcases. Delta debugging finds a *1-minimal* subset, an interesting subset such that removing any single element from it prevents it from being interesting. We use the fitness function as our notion of interesting and take the DIFFX operations as the set to minimize. On the 32-item "ultrix look" script, delta debugging performs only 10 fitness evaluations and produces a 1-minimal patch (final `diff` size: 11 lines long); see Section 4.3 for an analysis.

4 Experiments

We report experiments and case studies designed to: **1) Evaluate performance and scalability** by finding repairs for multiple legacy programs; **2) Measure run-time cost** in terms of fitness function evaluations and elapsed time; **3) Evaluate the success rate** of the evolutionary search; and **4) Understand how testcases affect repair quality**, characterizing the solutions found by our technique.

4.1 Experimental Setup

We selected several open source benchmarks from several domains, all of which have known defects. These include benchmarks taken from Miller *et al.*'s work on *fuzz*

Program	Version	LOC	Statements	Program Description	Fault
`gcd`	example	22	10	example from Section 2	infinite loop
`uniq`	ultrix 4.3	1146	81	duplicate text processing	segfault
`look`	ultrix 4.3	1169	90	dictionary lookup	segfault
`look`	svr4.0 1.1	1363	100	dictionary lookup	infinite loop
`units`	svr4.0 1.1	1504	240	metric conversion	segfault
`deroff`	ultrix 4.3	2236	1604	document processing	segfault
`nullhttpd`	0.5.0	5575	1040	webserver	remote heap buffer exploit
`indent`	1.9.1	9906	2022	source code processing	infinite loop
`flex`	2.5.4a	18775	3635	lexical analyzer generator	segfault
`atris`	1.0.6	21553	6470	graphical tetris game	local stack buffer exploit
total		63249	15292		

Figure 4. Benchmark programs used in our experiments, with size in lines of code (LOC). The 'Statements' column gives the number of applicable statements as defined in Section 3.2.

testing, in which programs crash when given random inputs [22]. The nullhttpd[1] and atris[2] benchmarks were taken from public vulnerability reports.

Testcases. For each program, we used a single negative testcase that elicits the fault listed in Figure 4. No special effort was made in choosing the negative testcase. For example, for the fuzz testing programs, we selected the first fuzz input that evinced a fault. A small number (e.g., 2–6) of positive testcases was selected for each program. In some cases, we used only non-crashing fuzz inputs as testcases; in others we manually created simple positive testcases. Testcase selection is an important topic, and in Section 4.3 we report initial results on how it affects repair quality.

Parameters. Our algorithms have several global weights and parameters that potentially have large impact on system performance. A systematic study of parameter values is beyond the scope of this paper. We report results for one set of parameters that seemed to work well across most of the application examples we studied. We chose pop_size $= 40$, which is small compared to other GP projects; on each trial, we ran the GP for a maximum of ten generations (also a small number), and we set $W_{PosT} = 1$ and $W_{NegT} = 10$. With the above parameter settings fixed, we experimented with two parameter settings for W_{Path} and W_{mut}:

$$\{W_{Path} = 0.01, W_{mut} = 0.06\}$$
$$\{W_{Path} = 0.00, W_{mut} = 0.03\}$$

Note that $W_{Path} = 0.00$ means that if a statement appears on a positive testcase path then it will not be considered for mutation, and $W_{Path} = 0.01$ means such statements will be considered infrequently.

The *weighted path length* is the weighted sum of statements on the negative path, where statements also on the positive path receive a weight of $W_{Path} = 0.01$ and statements only on the negative path receive a weight of 1.0. This roughly estimates the complexity of the search space and is correlated with algorithm performance (Section 4.4).

We define one *trial* to consist of at most two serial invocations of the GP loop using the parameter sets above in order. We stop the trial if an initial repair is discovered. We performed 100 random trials for each program, reporting the fraction of successes and the time to find the repair.

Optimizations. When calculating fitness, we memoize fitness results based on the pretty-printed abstract syntax tree. Thus two variants with different abstract syntax trees that yield the same source code are not evaluated twice. Similarly, variants that are copied without change to the next generation are not reevaluated. Beyond this caching, the prototype tool is not optimized; for example, it makes a deep copy of the AST before performing crossover and mutation. Incremental compilation approaches and optimizations are left as future work.

4.2 Experimental Results

Figure 5 summarizes experimental results for ten C programs. Successful repairs were generated for each program. The 'Initial Repair' heading reports timing information for the genetic programming phase and does not include the time for repair minimization.

The 'Time' column reports the wall-clock average time required for a trial that produced a primary repair. Our experiments were conducted on a quad-core 3 GHz machine; with a few exceptions, the process was CPU-bound. The GP prototype is itself single-threaded, with only one fitness evaluation at a time, during a fitness evaluation we execute all testcases in parallel. The 'fitness' column lists the average number of fitness evaluations performed during a successful trial. Fitness function evaluation is typically the

[1] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2002-1496
[2] http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=290230

Program	LOC	Positive Tests	$\|Path\|$	Initial Repair Time	Initial Repair fitness	Initial Repair Success	Initial Repair Size	Minimized Repair Time	Minimized Repair fitness	Minimized Repair Size
`gcd`	22	5x human	1.3	149 s	41.0	54%	21	4 s	4	2
`uniq`	1146	5x fuzz	81.5	32 s	9.5	100%	24	2 s	6	4
`look-u`	1169	5x fuzz	213.0	42 s	11.1	99%	24	3 s	10	11
`look-s`	1363	5x fuzz	32.4	51 s	8.5	100%	21	4 s	5	3
`units`	1504	5x human	2159.7	107 s	55.7	7%	23	2 s	6	4
`deroff`	2236	5x fuzz	251.4	129 s	21.6	97%	61	2 s	7	3
`nullhttpd`	5575	6x human	768.5	502 s	79.1	36%	71	76 s	16	5
`indent`	9906	5x fuzz	1435.9	533 s	95.6	7%	221	13 s	13	2
`flex`	18775	5x fuzz	3836.6	233 s	33.4	5%	52	7 s	6	3
`atris`	21553	2x human	34.0	69 s	13.2	82%	19	11 s	7	3
average			881.4	184.7 s	36.9	58.7%	53.7	12.4 s	8.0	4.0

Figure 5. Experimental Results on 63249 lines of code: The 'Positive Tests' column describes the positive tests (Section 3.4). The '$|Path|$' columns give the weighted path length. 'Initial Repair' gives the average performance for one trial, in terms of 'Time' (the average time of each successful trial, including compilation time and testcase evaluation), 'fitness' (the average number of fitness evaluations in a successful trial), 'Success' (how many of the random trials resulted in a repair). 'Size' reports the average `diff` size between the original source and the primary repair, in lines. 'Minimized Repair' reports the same information but describes the process of producing a 1-minimal repair from the first initial repair found; the minimization process is deterministic and always succeeds.

dominant expense in GP applications as problem size increases, which is why we record number of fitness function evaluations. An average successful trial terminates in 184.7 seconds after 36.9 fitness evaluations. Of that time, 54% is spent executing testcases (i.e., in the fitness function) and another 30% is spent compiling program variants.

The 'Success' column gives the fraction of trials that were successful. On average, over half of the trials produced a repair, although most of the benchmarks either succeeded very frequently or very rarely. Low success rates can be mitigated by running multiple independent trials in parallel. On average there were 5.5 insertions, deletions and swaps applied to a variant between generations: a single application of our mutation operator is comparable to several standard GP mutations. The average initial repair was evolved using 3.5 crossovers and 1.8 mutations over 6.0 generations. Section 4.4 discusses success in detail.

The 'Size' column lists the size, in lines, of the primary repair. Primary repairs typically contain extraneous changes (see Section 3.5). The 'Minimized Repair' heading gives performance information for producing a 1-minimal patch that passes all of the testcases. Minimization is deterministic and takes fewer seconds and fitness evaluations. The final minimized patch is quite manageable, averaging 4 lines.

4.3 Repair Quality and Testcases

In some cases, the evolved and minimized repair is exactly as desired. For example, the minimized repair for `gcd` inserts a single `exit(0)` call, as shown in Section 2 (the other line in the two-line patch is location information explaining where to insert the call). Measuring the *quality* of a repair is both a quantitative and a qualitative notion: the repair must compile, fix the defect, and avoid compromising required functionality. All of the repairs compile, fix the defect, and avoid compromising required functionality in the positive testcases provided.

For `uniq`, the function `gline` reads user input into a static buffer using a temporary pointer without bounds checks; our fix changes the increment to the temporary pointer.

For ultrix `look`, our repair changes the handling of command-line arguments, avoiding a subsequent buffer overrun in the function `getword`, which reads user input into static buffer without bounds checks.

In svr4 `look`, the `getword` function correctly bounds-checks its input, so the previous segfault is avoided. However, a loop in `main` uses a buggy binary search to seek through a dictionary file looking for a word. If the dictionary file is not in sorted order, the binary search loop never terminates. Our patch adds a new exit condition to the loop.

In `units`, the function `convr` reads user input to a static buffer without bounds checks, and then passes a pointer to the buffer to `lookup`. Our repair changes the `lookup` function so that it calls `init` on failure, re-initializing data structures and avoiding the segfault.

In `deroff`, the function `regline` sometimes reads user input to a static buffer without bounds checks. `regline` calls `backsl` for escape sequence processing; our repair changes the handling of delimiters in `backsl`, preventing `regline` from segfaulting.

In `indent` program has an error in its handling of comments that leads to an infinite loop. Our repair removes handling of C comments that are not C++ comments. This removes the infinite loop but reduces functionality.

In `flex`, the `flexscan` function calls `strcpy` from the `yytext` pointer into a static buffer in seven circumstances. In some, `yytext` holds controlled input fragments; in others, it points to unterminated user input. Our patch changes one of the uncontrolled copies.

In `atris`, the `main` function constructs the file path to the user's preference file by using `sprintf` to concatenate the value of the `HOME` environment variable with a file name into a static buffer. Our repair removes the `sprintf` call, leaving all users with the default global preferences.

In `nullhttpd` there is a remote exploitable heap buffer overflow in `ReadPOSTData`: the user-supplied `POST` length value is trusted, and negative values can be used to copy user-supplied data into arbitrary memory locations. We used six positive testcases: `GET index.html`, `GET blank.html`, `GET notfound.html`, `GET icon.gif`, `GET directory/`, and `POST recopy-posted-value.pl`. Our generated repair changes `read_header` so that `ReadPOSTData` is not called. Instead, the processing in `cgi_main` is used, which invokes `write` to copy the `POST` data only after checking if `in_ContentLength > 0`.

To study the importance of testcases, we ran our `nullhttpd` experiment without the `POST` testcase. Our algorithm generates a repair that disables `POST` functionality; all `POST` requests generate an HTML bad request error reply. As a quick fix this is not unreasonable, and is safer than the common alarm practice of running in read-only mode.

Our repair technique aggressively prunes functionality to repair the fault unless that functionality is guarded by testcases. For `indent` we remove C comment handling without a C testcase. For `atris` we remove handling of local preference files. Our technique also rarely inserts local bounds checks directly, instead favoring higher-level control-flow changes that avoid the problem, as in `units` or `deroff`.

Our approach thus presents a tradeoff between rapid repairs that address the fault and using more testcases to obtain a more human repair. In the `atris` example, a security vulnerability in a 20000 line program is repaired in under 100 seconds using only two testcases and a minimal sacrifice of non-core functionality. In the `nullhttpd` example, a similar vulnerability is fully repaired in 10 minutes, with all relevant functionality retained. Time-aware test suite prioritization techniques exist for choosing a useful and quick-to-execute subset of a large test suite [26, 29]; in our experiments five testcases serve as a reasonable lower bound. We do not view the testcase requirement for our algorithm as a burden, especially compared to techniques that require formal specifications. As the `nullhttpd` example shows, if the repair sacrifices functionality that was not in the positive test suite, a new repair can be made from more test cases.

4.4 Success and Sensitivity

We observed a high variance in success rates between programs. GP is ultimately a heuristic-guided random search; the success rate in some sense measures the difficulty of finding the solution. A high success rate indicates that almost any random choice or coin toss will hit upon the solution; a low success rate means that many circumstances must align for us to find a repair. Without our weighted path representation, the success rate would decrease with program size as more and more statements would have to be searched to find the repair.

In practice, the success rate seems more related to the structure of the program and the location of the fault than to the nature of the fault. For example, in our experiments infinite loops have an average success rate of 54% while buffer overruns have 61% — both very similar to the overall average of 59%. Instead, the success rate is inversely related to the weighted path length. The weighted path length loosely counts statements on the negative path that are not also on the positive path; our genetic operators are applied along the weighted path. The weighted path is thus the haystack in which we search for needles. For example, `flex` has the longest weighted path and also the lowest success rate; in practice its repair is found inside the `flexscan` function, which is over 1,400 lines of machine-generated code implementing a DFA traversal via a huge `switch` statement. Finding the right `case` requires luck. The `indent` and `units` cases are analogous. Note that adding additional positive testcases actually reduces the weighted path length, and thus improves the success rate, although it also increases the runtime. In addition, we can use existing *path slicing* tools; Jhala and Majumdar, for example, slice a 82,695-step negative path on the `gcc` Spec95 benchmark to 43 steps [18].

Finally, the parameter set $W_{Path} = 0.01$ and $W_{mut} = 0.06$ works well in practice; more than 60% of the successful trials are from these settings. Our weighted path representation is critical to success; without weighting from positive testcases our algorithm rarely succeeds (e.g., `gcd` fails 100% of the time). We have also tried higher mutation chances and note that success worsens beyond 0.12.

4.5 Limitations, Threats to Validity

We assume that the defect is reproducible and that the program behaves deterministically on the testcases. This limitation can be mitigated by running the testcases multiple times, but ultimately if the program behavior is random we may make an incorrect patch. We further assume that positive testcases can encode program requirements. Testcases are much easier to obtain than formal specifications or code annotations, but if too few are used, the repair may

sacrifice important functionality. In practice too many testcases may be available, and a large number will slow down our technique and constrain the search space. We further assume that the path taken along the negative testcase is different from the positive path. If they overlap completely our weighted representation will not be able to guide GP modifications. Finally, we assume that the repair can be constructed from statements already extant in the program; in future work we plan to use a library of repair templates.

5 Related Work

Our approach automatically repairs programs without specifications. In previous work we developed an automatic algorithm for soundly repairing programs with specifications [31]. The previous work suffers from three key drawbacks. First, while it is sound with respect to the given policy, it may generate repairs that sacrifice other required functionality. In this paper, a sufficient set of positive testcases prevents us from generating such harmful repairs. Second, the previous work only repairs single-threaded violations of temporal safety properties; it cannot handle multithreaded programs or liveness properties, such as the three infinite loop faults handled in this paper. Third, the previous work requires a formal specification of the policy being violated by the fault. In practice, despite recent advances in specification mining (e.g., [20]), formal specifications are rarely available. For example, there were no formal specifications available for any of the programs repaired here.

Trace localization [8], minimization [15], and explanation [10] projects also aim to elucidate faults and ease repairs. These approaches typically narrow down a large counterexample backtrace (the error symptom) to a few lines (a potential cause). Our work improves upon this in three fundamental ways. First, a narrowed trace or small set of program lines is not a concrete repair. Second, our approach works with any detected fault, not just those found by static analysis tools that produce counterexamples. Finally, their algorithms are limited to the given trace and source code and will thus never localize the "cause" of an error to a missing statement or suggest that a statement be moved. Our approach can infer new code that should be added or swapped: five of our ten minimized repairs required insertions or swaps. Their work could also be viewed as complementary to ours; a defect found by static analysis might be repaired and explained automatically, and both the repair and the explanation could be presented to developers.

Demsky *et al.* [11] present a technique for data structure repair. Given a formal specification of data structure consistency, they modify a program so that if the data structures ever become inconsistent, they can be modified back to a consistent state at runtime. Their technique does not modify the program source code in a user-visible way. Instead, it inserts run-time monitoring code that "patches up" inconsistent state so that the buggy program can continue to execute. Data structure repair requires formal specifications, which can either be provided by a developer or automatically inferred. There are, however, issues that may make it difficult or impossible to infer adequate specifications in practice. As a result, it may be difficult to evaluate the quality of the repair and subject programs continue to incur the run-time overhead. Finally, their technique only targets errors that corrupt data structures — it is not designed to address the full range of logic errors. The `gcd` infinite loop in Section 2, for example, is outside the scope of this technique. Our techniques are complementary: in cases where runtime data structure repair does not provide a viable long-term solution, it may enable the program to continue to execute while our technique searches for a long-term repair.

Rinard *et al.* [25] present a notion of *failure-oblivious computing*, in which compiler-inserted bounds checks discard invalid memory accesses, allowing servers to avoid security attacks. Boundless memory blocks store out of bounds writes for subsequent retrieval for corresponding out of bounds reads. Techniques also exist for eliminating infinite loops. The overhead of these approaches varies (from 3% to 8x), memory errors are the primary consideration, and a localized patch is not produced. However, the approach has been applied automatically and successfully to large server programs.

Arcuri [6, 7] proposed the idea of using GP to repair software bugs automatically, but our work is the first to report substantial experimental results on real programs with real bugs. The approaches differ in several details: we use execution paths to localize evolutionary search operators, and we do not rely on a formal specification in the fitness evaluation phase. Where they control "code bloat" along the way, we minimize our high-fitness solution after the evolutionary search has finished.

The field of Search-Based Software Engineering (SBSE) [17] uses evolutionary and related methods for software testing, e.g., to develop test suites [29, 30]. SBSE also uses evolutionary methods to improve software project management and effort estimation [9], to find safety violations [3], and in some cases to re-factor or re-engineer large software bases [28]. In SBSE, most innovations in the GP technique involve new kinds of fitness functions, and there has been less emphasis on novel representations and operators, such as those we explored in this paper.

6 Conclusions

We present a fully automated technique for repairing bugs in off-the-shelf legacy software. Instead of using formal specifications or special coding practices, we use an extended form of genetic programming to evolve program

variants. We consider only certain classes of repairs, using one part of a program as a template to repair another part. Our GP algorithm uses a representation that combines abstract syntax trees with weighted violating paths; these insights allow our search to scale to large programs. We use standard testcases to show the fault and to encode required functionality; our initial repair is a variant that passes all testcases. The initial repair is minimized using delta debugging and structural differencing algorithm. We are able to generate and minimize repairs for ten different C programs totalling 63,000 lines in under 200 seconds on average.

Acknowledgments

We thank David E. Evans, John C. Knight, Anh Nguyen-Tuong, and Martin Rinard for insightful discussions.

References

[1] 36 human-competitive results produced by genetic programming. `http://www.genetic-programming.com/humancompetitive.html`, Aug. 17, 2008.

[2] R. Al-Ekram, A. Adma, and O. Baysal. diffX: an algorithm to detect changes in multi-version XML documents. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.

[3] E. Alba and F. Chicano. Finding safety errors with ACO. In *Conference on Genetic and Evolutionary Computation*, pages 1066–1073, 2007.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[5] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[6] A. Arcuri. On the automation of fixing software bugs. In *Proceedings of the Doctoral Symposium of the IEEE International Conference on Software Engineering*, 2008.

[7] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, 2008.

[8] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003.

[9] A. Barreto, M. de O. Barros, and C. M. Werner. Staffing a software project: a constraint satisfaction and optimization-based approach. *Computers and Operations Research*, 35(10):3073–3089, 2008.

[10] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering*, pages 73–82, 2004.

[11] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis*, pages 233–244, 2006.

[12] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[13] 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*, pages 57–72, 2001.

[14] S. Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261:872–878, Aug. 13 1993.

[15] A. Groce and D. Kroening. Making the most of BMC counterexamples. In *Electronic Notes in Theoretical Computer Science*, volume 119, pages 67–81, 2005.

[16] S. Gustafson, A. Ekart, E. Burke, and G. Kendall. Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Machines*, pages 271–290, 2004.

[17] M. Harman. The current state and future of search based software engineering. In *International Conference on Software Engineering*, pages 342–357, 2007.

[18] R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation*, pages 38–47, 2005.

[19] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[20] C. Le Goues and W. Weimer. Specification mining with few false positives. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming language design and implementation*, pages 141–154, 2003.

[22] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[23] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: An infrastructure for C program analysis and transformation. In *International Conference on Compiler Construction*, pages 213–228, Apr. 2002.

[24] C. V. Ramamoothy and W.-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.

[25] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Operating Systems Design and Implementation*, pages 303–316, 2004.

[26] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[27] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[28] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Conference on Genetic and Evolutionary Computation*, pages 1909–1916, 2006.

[29] K. Walcott, M. Soffa, G. Kapfhammer, and R. Roos. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2006.

[30] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Conference on Genetic and Evolutionary Computation*, pages 1925–1932, 2006.

[31] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[32] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.