GenProg: A Generic Method for Automatic Software Repair

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, *Senior Member*, *IEEE*, and Westley Weimer

Abstract—This paper describes GenProg, an automated method for repairing defects in off-the-shelf, legacy programs without formal specifications, program annotations, or special coding practices. GenProg uses an extended form of genetic programming to evolve a program variant that retains required functionality but is not susceptible to a given defect, using existing test suites to encode both the defect and required functionality. Structural differencing algorithms and delta debugging reduce the difference between this variant and the original program to a minimal repair. We describe the algorithm and report experimental results of its success on 16 programs totaling 1.25 M lines of C code and 120K lines of module code, spanning eight classes of defects, in 357 seconds, on average. We analyze the generated repairs qualitatively and quantitatively to demonstrate that the process efficiently produces evolved programs that repair the defect, are not fragile input memorizations, and do not lead to serious degradation in functionality.

Index Terms—Automatic programming, corrections, testing and debugging.

1 Introduction

Software quality is a pernicious problem. Mature software projects are forced to ship with both known and unknown bugs [1] because the number of outstanding software defects typically exceeds the resources available to address them [2]. Software maintenance, of which bug repair is a major component [3], [4], is time-consuming and expensive, accounting for as much as 90 percent of the cost of a software project [5] at a total cost of up to $70 billion per year in the US [6], [7]. Put simply: Bugs are ubiquitous, and finding and repairing them are difficult, time-consuming, and manual processes.

Techniques for automatically detecting software flaws include intrusion detection [8], model checking and lightweight static analyses [9], [10], and software diversity methods [11], [12]. However, detecting a defect is only half of the story: Once identified, a bug must still be repaired. As the scale of software deployments and the frequency of defect reports increase [13], some portion of the repair problem must be addressed automatically.

This paper describes and evaluates Genetic Program Repair ("GenProg"), a technique that uses existing test cases to automatically generate repairs for real-world bugs in off-the-shelf, legacy applications. We follow Rinard et al. [14] in defining a *repair* as a patch consisting of one or more code changes that, when applied to a program, cause it to pass a set of test cases (typically including both tests of required behavior as well as a test case encoding the bug). The test cases may be human written, taken from a regression test suite, steps to reproduce an error, or generated automatically. We use the terms "repair" and "patch" interchangeably. GenProg does not require formal specifications, program annotations, or special coding practices. GenProg's approach is generic, and the paper reports results demonstrating that GenProg can successfully repair several types of defects. This contrasts with related approaches which repair only a specific type of defect (such as buffer overruns [15], [16]).

GenProg takes as input a program with a defect and a set of test cases. GenProg may be applied either to the full program source or to individual modules. It uses *genetic programming* (GP) to search for a program variant that retains required functionality but is not vulnerable to the defect in question. GP is a stochastic search method inspired by biological evolution that discovers computer programs tailored to a particular task [17], [18]. GP uses computational analogs of biological mutation and crossover to generate new program variations, which we call *variants*. A user-defined *fitness function* evaluates each variant; GenProg uses the input test cases to evaluate the fitness, and individuals with high fitness are selected for continued evolution. This GP process is successful when it produces a variant that passes all tests encoding the required behavior and does not fail those encoding the bug. Although GP has solved an impressive range of problems (e.g., [19]), it has not previously been used either to evolve off-the-shelf legacy software or to patch real-world vulnerabilities, despite various proposals directed at automated error repair, e.g., [20].

A significant impediment for GP efforts to date has been the potentially infinite space that must be searched to find a correct program. We introduce three key innovations to address this longstanding problem [21]. First, GenProg operates at the *statement level* of a program's abstract syntax tree (AST), increasing the search granularity. Second, we hypothesize that a program that contains an error in one area likely implements the correct behavior elsewhere [22]. Therefore, GenProg uses only statements from the program

- *C. Le Goues and W. Weimer are with the Department of Computer Science, University of Virginia, 85 Engineer's Way, PO Box 400740, Charlottesville, VA 22904-4740. E-mail: {legoues, weimer}@cs.virginia.edu.*
- *T. Nguyen and S. Forrest are with the Department of Computer Science, University of New Mexico, MSC01 1130, 1 University of New Mexico, Albuquerque, NM 87131-0001. E-mail: {tnguyen, forrest}@cs.unm.edu.*

Manuscript received 16 Mar. 2010; revised 6 Oct. 2010; accepted 21 Sept. 2011; published online 30 Sept. 2011.
Recommended for acceptance by J.M. Atlee and P. Inverardi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2010-03-0078.
Digital Object Identifier no. 10.1109/TSE.2011.104.

```
1  char* ProcessRequest() {
2    int length, rc;
3    char[10] request_method;
4    char* buff;
5    while(line = sgets(line, socket)) {
6      if(line == "Request:")
7        strcpy(request_method, line+12)
8      if(line == "Content-Length:")
9        length=atoi(line+16);
10   }
11   if (request_method == "GET")
12     buff=DoGETRequest(socket,length);
13   else if(request_method == "POST") {
14     buff=calloc(length, sizeof(char));
15     rc=recv(socket,buff,length)
16     buff[length]='\0';
17   }
18   return buff;
19 }
```

(a) Buggy webserver code snippet.

```
4    ...
5    while(line = sgets(line, socket)) {
6      if(line == "Request:")
7        strcpy(request_method, line+12)
8      if(line == "Content-Length:")
9        length=atoi(line+16);
10   }
11   if (request_method == "GET")
12     buff=DoGETRequest(socket,length);
13   else if(request_method == "POST") {
14 +   if (length <= 0)
15 +       return null;
16     buff=calloc(length, sizeof(char));
17     rc=recv(socket,buff,length)
18     buff[length]='\0';
19   }
20   return buff;
21 }
```

(b) Patched webserver.

Fig. 1. Pseudocode of a buggy webserver implementation, and a repaired version of the same program.

itself to repair errors and does not invent new code. Finally, GenProg localizes genetic operators to statements that are executed on the failing test case. This third point is critical: Fault localization is, in general, a hard and unsolved problem. The scalability of our approach relies on existing, imperfect strategies, and there exist classes of defects (e.g., nondeterministic bugs) which cannot always be localized. For the defects considered here, however, we find that these choices reduce the search space sufficiently to permit the automated repair of a varied set of both programs and errors.

The GP process often introduces irrelevant changes or dead code along with the repair. GenProg uses structural differencing [23] and delta debugging [24] in a postprocessing step to obtain a 1-minimal set of changes to the original program that permits it to pass all of the test cases. We call this set the *final repair*.

The main contributions of this paper are:

- GenProg, an algorithm that uses GP to automatically generate patches for bugs in programs, as validated by test cases. The algorithm includes a novel and efficient representation and set of operations for applying GP to this domain. This is the first work to demonstrate the use of GP to repair software at the scale of real, unannotated programs with publicly documented bugs.
- Experimental results showing that GenProg can efficiently repair errors in 16 C programs. Because the algorithm is stochastic, we report success rates for each program averaged over 100 trials. For every program, at least one trial found a successful repair, with the average success rates ranging from 7 to 100 percent. Across all programs and all trials, we report an average success rate of 77 percent.
- Exerimental results demonstrating that the algorithm can repair multiple types of errors in programs drawn from multiple domains. The errors span eight different defect types: infinite loop, segmentation fault, remote heap buffer overflow to inject code, remote heap buffer overflow to overwrite variables, nonoverflow denial of service, local stack buffer overflow, integer overflow, and format string vulnerability. The benchmark programs include Unix utilities, servers, media players, text processing programs, and games. The 16 benchmarks total over 1.25 M lines of code (LOC), although GenProg operates directly on 120K lines of program or module code.

Some of these points were previously presented in early versions of this work [25], [26] or summarized for general audiences [27]. This paper extends those results to include:

- **New repairs.** Previous work showed repairs on 11 programs totaling 63K lines of code and four classes of errors. We present five additional programs, and show that GenProg can operate on both an entire program's source code as well as at the module level. The new benchmarks consist of 1.2M new lines of source code, 60K new lines of repaired code (either module or whole program), and four new types of errors, a significant increase that substantiates GenProg's ability to scale to real-world systems.
- **Closed-loop repair.** A description and proof-of-concept evaluation of a closed-loop repair system that integrates GenProg with anomaly intrusion detection.
- **Repair quality.** A partial evaluation of the quality of the produced repairs, first manually and then quantitatively, using indicative workloads, fuzz testing, and variant bug-inducing input. Our preliminary findings suggest that the repairs are not fragile memorizations of the input, but instead address the defect while retaining required functionality.

2 Motivating Example

In this section, we use an example defect to highlight the important insights underlying the GenProg approach and to motivate important design decisions.

Consider the pseudocode shown in Fig. 1a, adapted from a remote-exploitable heap buffer overflow vulnerability in the **`nullhttpd v0.5.0`** webserver. Function **`Process Request`** processes an incoming request based on data copied from the request header. Note that on line 14, the call to **`calloc`** to allocate memory to hold request contents trusts

the content length provided by a POST request, as copied from the header on line 8. A malicious attacker can provide a negative value for **Content-Length** and a malicious payload in the request body to overflow the heap and kill or remotely gain control of the running server.

To automatically repair this program, we must first codify desired behavior. For example, we can write a test case that sends a POST request with a negative content length and a malicious payload to the webserver, and then checks the webserver to determine if it is still running. Unmodified **nullhttpd** fails this test case.

At a high level, GenProg searches for valid variants of the original program that do not display the specified buggy behavior. However, searching randomly through related programs may yield undesirable results. Consider the following variant:

```
1  char* ProcessRequest() { return null; }
```

This version of **ProcessRequest** does not crash on the bug-encoding test case, but also fails to process any requests at all. The repaired program should pass the error-encoding test case while retaining core functionality. Such functionality can also be expressed with test cases, such as a standard regression test that obtains **index.html** and compares the retrieved copy against the expected output.[1]

To satisfy these goals, program modifications should ideally focus on regions of code that affect the bad behavior without affecting the good behavior. We therefore employ a simple *fault localization* strategy to reduce the search space. We instrument the program to record all lines visited when processing the test cases, and favor changes to locations that are visited exclusively by the negative test case. The standard regression test visits lines 1-12 and 18 (and lines in **DoGETRequest**). The test case demonstrating the error visits lines 1-11 and 13-18. Mutation and crossover operations are therefore focused on lines 13-17, which exclusively implement POST functionality.

Despite this fault localization, there are still many possible changes to explore. To further constrain the search, we assume that most defects can be repaired by adapting existing code from another location in the program. In practice, a program that makes a mistake in one location often handles a similar situation correctly in another [22]. This hypothesis is correct for **nullhttpd**. Although the POST request handling in **ProcessRequest** does not do a bounds check on the user-specified content length, the **cgi_main** function, implemented elsewhere, does:

```
502  if (length <= 0) return null;
```

Fault localization biases the modifications toward POST request code. The restriction to use only existing code for insertions further limits the search, and eventually GenProg tries inserting the check from **cgi_main** into **ProcessRequest**, shown in Fig. 1b. A program with this version of **ProcessRequest** passes both test cases; we call it the *primary repair*. GP can produce spurious changes in addition to those that repair the program; for example, the search might have randomly inserted **return DoGet Request(socket,length)** at line 22, after the original **return**. This insertion is not dangerous because it will never be executed, but it does not contribute to the repair. We remove such extraneous changes in a postprocessing step. The resulting minimal patch is the *final repair*; we present it in traditional **diff** format.

We formalize this procedure and describe concrete implementation details in the next section.

1. In practice, we use several test cases to express program requirements; we describe only one here for brevity.

Input: Program P to be repaired.
Input: Set of positive test cases $PosT$.
Input: Set of negative test cases $NegT$.
Input: Fitness function f.
Input: Variant population size pop_size.
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: $\quad Viable \leftarrow \{\langle P, Path_P \rangle \in Popul \mid f(P) > 0\}$
7: $\quad Popul \leftarrow \emptyset$
8: $\quad NewPop \leftarrow \emptyset$
9: $\quad$ **for all** $\langle p_1, p_2 \rangle \in \mathsf{select}(Viable, f, \mathsf{pop\_size}/2)$ **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** $\langle V, Path_V \rangle \in NewPop$ **do**
14: $\quad\quad Popul \leftarrow Popul \cup \{\mathsf{mutate}(V, Path_V)\}$
15: $\quad$ **end for**
16: **until** $f(V) = \mathsf{max\_fitness}$ for some V contained in $Popul$
17: **return** $\mathsf{minimize}(V, P, PosT, NegT)$

Fig. 2. High-level pseudocode for GenProg. Lines 5-16 describe the GP search for a feasible variant. Subroutines such as $\mathrm{mutate}(V, Path_V)$ are described subsequently.

3 Technical Approach

Fig. 2 gives pseudocode for GenProg. GenProg takes as input source code containing a defect and a set of test cases, including a failing *negative* test case that exercises the defect and a set of passing *positive* test cases that describe requirements. The GP maintains a population of program variants represented as trees. Each variant is a modified instance of the original defective program; the modifications are generated by the mutation and crossover operations, described in Section 3.2. The call to initial_population on line 4 uses mutation operators to construct an initial GP population based on the input program and test cases. A *fitness function* evaluates each individual's *fitness*, or desirability. GenProg uses the input test cases to guide the GP search (lines 1-3 of Fig. 2, Section 3.1) as well as to evaluate fitness (Section 3.3). A GP iterates by *selecting* high-fitness individuals to copy into the next generation (line 9, Section 3.2) and introducing variations with the mutation and crossover operations (lines 13-15 and line 10). This cycle repeats until a goal is achieved—a variant is found that passes all the test cases—or a predetermined resource limit is consumed. Finally, GenProg minimizes the successful variant (line 17, Section 3.4)

3.1 Program Representation

GenProg represents each variant (candidate program) as a pair:

1. An *abstract syntax tree* that includes all of the statements in the program.
2. A *weighted path* consisting of a list of program statements, each associated with a weight based on that statement's occurrence in various test case execution traces.

GenProg generates a program AST using the off-the-shelf CIL toolkit [28]. ASTs express program structure at multiple levels of abstraction or granularity. GenProg operates on the constructs that CIL defines as *statements*, which includes all assignments, function calls, conditionals, blocks, and looping constructs. GenProg does not directly modify *expressions* such as "`(1-2)`" or "`(!p)`" nor does it ever directly modify low-level control-flow directives such as **`break`**, **`continue`**, or **`goto`**. This genotype representation reflects a tradeoff between expressive power and scalability. Because of these constraints on permitted program modifications, the GP never generates syntactically ill-formed programs (e.g., it will never generate unbalanced parentheses). However, it can generate variants that fail to compile due to a semantic error by, for example, moving the use of a variable out of scope.

The *weighted path* is a sequence of $\langle \text{statement}, \text{weight} \rangle$ pairs that constrains the mutation operators to a small, likely relevant (more highly weighted) subset of the program tree. Statements not on the weighted path (i.e., with weight 0) are never modified, although they may be copied into the weighted path by the mutation operator (see Section 3.2). Each new variant has the same number of pairs and the same sequence of weights in its weighted path as the original program. This is necessary for the crossover operation (described below).

To construct the weighted path, we apply a transformation that assigns each statement a unique number and inserts code to log an event (visit) each time the statement is executed (lines 1-2 of Fig. 2). Duplicate statements are removed from the list: That is, we do not assume that a statement visited frequently (e.g., in a loop) is likely to be a good repair site. However, we do respect statement order (determined by the first time a statement is visited), so the weighted path is a sequence, rather than a set. Any statement visited during the execution of a negative test case is a candidate for repair, and its initial weight is set to 1.0. All other statements are assigned a weight of 0.0 and never modified. The initial weights of the statements on the negative test case execution path are modified further by changing the weights of those statements that were also executed by a positive test case. The goal is to bias the modifications toward portions of the source code that are likely to affect the bad behavior, while avoiding those that influence good behavior. $\text{set\_weights}(Path_{NegT}, Path_{PosT})$ on line 3 of Fig. 2 sets the weight of every path statement that is visited during at least one positive test case to a parameter W_{Path}. Choosing $W_{Path} = 0$ prevents modification of any statement visited during a positive test case by removing it from the path; we found that values such as $W_{Path} = 0.01$ typically work better in practice.

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⟩
```

Fig. 3. The mutation operator. Updates to $Path_P$ also update the AST P.

The weighted path serves to localize the fault. This fault localization strategy is simple, and by no means state of the art, but has worked in practice for our benchmark programs. We do not claim any new results in fault localization, and instead view it as an advantage that we can use relatively off-the-shelf approaches. Path weighting is necessary to repair the majority of the programs we have investigated: Without it, the search space is typically too large to search efficiently. However, effective fault localization for both automatic and manual repair remains a difficult and unsolved problem, and there exist certain types of faults which remain difficult to impossible to localize. We expect that GenProg will improve with advances in fault localization, and leave the extension of the technique to use more sophisticated localization methods as future work.

3.2 Selection and Genetic Operators

Selection. The code on lines 6-9 of Fig. 2 implements the process by which GenProg selects individual variants to copy over to the next generation. GenProg discards individuals with fitness 0 (variants that do not compile or that pass no test cases) and places the remainder in $Viable$ on line 6. It then uses a selection strategy to select $\text{pop\_size}/2$ members of a new generation from the previous iteration; these individuals become the new mating pool. We have used both *stochastic universal sampling* [29], in which each individual's probability of selection is directly proportional to its relative fitness f, and *tournament selection* [30], where small subsets of the population are selected randomly (a tournament) and the most fit member of the subset is selected for the next generation. This process is iterated until the new population is selected. Both selection techniques produce similar results in our application.

Two GP operators, mutation and crossover, create new variants from this mating pool.

Mutation. Fig. 3 shows the high-level pseudocode for the *mutation* operator. Mutation has a small chance of changing any particular statement along the weighted path (line 1). Changes to statements in $Path_P$ are reflected in its corresponding AST P. A statement is mutated with

Input: Parent programs P and Q.
Input: Paths $Path_P$ and $Path_Q$.
Output: Two new child program variants C and D.
1: $cutoff \leftarrow \mathsf{choose}(|Path_P|)$
2: $C, Path_C \leftarrow \mathsf{copy}(P, Path_P)$
3: $D, Path_D \leftarrow \mathsf{copy}(Q, Path_Q)$
4: **for** $i = 1$ to $|Path_P|$ **do**
5: **if** $i > cutoff$ **then**
6: $Path_C[i] \leftarrow Path_Q[i]$
7: $Path_D[i] \leftarrow Path_P[i]$
8: **end if**
9: **end for**
10: **return** $\langle C, Path_C\rangle$,$\langle D, Path_D\rangle$

Fig. 4. The crossover operator. Updates to $Path_C$ and $Path_D$ update the ASTs C and D.

probability equal to its weight, with the maximum number of mutations per individual determined by the global mutation rate (the parameter W_{mut}, set to 0.06 and 0.03 in our experiments; see Section 5.1). Line 2 uses these probabilities to determine if a statement will be mutated.

In genetic algorithms, mutation operations typically involve single bit flips or simple symbolic substitutions. Because our primitive unit is the statement, our mutation operator is more complicated, and consists of either a deletion (the entire statement is deleted), an insertion (another statement is inserted after it), or a swap with another statement. We choose from these options with uniform random probability (line 3). In the case of an insertion or swap, a second statement $stmt_j$ is chosen uniformly at random from anywhere in the program (lines 5 and 8), not just along the weighted path; a statement's weight does not influence the probability that it is selected as a candidate repair. This reflects our intuition about related changes: A program missing a null check probably includes one somewhere, but not necessarily on the negative path. In a swap, $stmt_i$ is replaced by $stmt_j$, while at the same time $stmt_j$ is replaced by $stmt_i$. We insert by transforming $stmt_i$ into a block statement that contains $stmt_i$ followed by $stmt_j$. In the current implementation, $stmt_j$ is not modified when inserted, although we note that intermediate variants may fail to compile if code is inserted which references out-of-scope variables. Deletions transform $stmt_i$ into an empty block statement; a deleted statement may therefore be modified in a later mutation operation.

In all cases, the new statement retains the old statement weight to maintain the invariant of uniform path lengths and weights between program variants and because inserted and swapped statements may not come from the weighted path (and may thus have no initial weight of their own).

Crossover. Fig. 4 shows the high-level pseudocode for the *crossover* operator. Crossover combines the "first part" of one variant with the "second part" of another, creating offspring variants that combine information from two parents. The crossover rate is 1.0—every surviving variant in a population undergoes crossover, though a variant will only be the parent in one such operation per generation. 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. We have experimented with other crossover operators (e.g., a crossover biased by path weights and a crossover with the original program) and found that they give similar results to the one-point crossover shown here.

3.3 Fitness Function

The *fitness function* evaluates the acceptability of a program variant. Fitness provides a termination criterion for the search and guides the selection of variants for the next generation. Our fitness function encodes software requirements at the test case level: *negative* test cases encode the fault to be repaired, while *positive* test cases encode functionality that cannot be sacrificed. We compile the variant's AST to an executable program, and then record which test cases the executable passes. Each successful positive test is weighted by the global parameter W_{PosT}; each successful negative test is weighted by the global parameter W_{NegT}. The fitness function is thus simply the weighted sum

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

The weights W_{PosT} and W_{NegT} should be positive; we give concrete values in Section 5. A variant that does not compile has fitness zero. For full safety, the test case evaluations can be run in a virtual machine or similar sandbox with a time out. Since test cases validate repair correctness, test suite selection is an important consideration.

3.4 Repair Minimization

The search terminates successfully when GP discovers a *primary repair* that passes all test cases. Due to randomness in the mutation and crossover algorithms, the primary repair typically contains at least an order-of-magnitude more changes than are necessary to repair the program, rendering the repairs difficult to inspect for correctness. Therefore, GenProg minimizes the primary repair to produce the *final repair*, expressed as a list of edits in standard `diff` format. Defects associated with such patches are more likely to be addressed [31].

GenProg performs minimization by considering each difference between the primary repair and the original program and discarding every difference that does not affect the repair's behavior on any of the test cases. Standard `diff` patches encode concrete, rather than abstract syntax. Since concrete syntax is inefficient to minimize, we have adapted the DIFFX XML differencing algorithm [23] to work on CIL ASTs. Modified DIFFX generates a list of tree-structured edit operations, such as "move the subtree rooted at node X to become the Yth child of node Z." This encoding is typically shorter than the corresponding `diff` patch, and applying part of a tree-based edit never results in a syntactically ill-formed program, both of which make such patches easier to minimize.

The minimization process finds a subset of the initial repair edits from which no further elements can be dropped without causing the program to fail a test case (a *1-minimal subset*). A brute-force search through all subsets of the initial list of edits is infeasible. Instead, we use *delta debugging* [24] to efficiently compute the one-minimal subset, which is

Program	Lines of Code		Description	Fault
	Total	Module		
gcd	22	22	example	infinite loop
zune	28	28	example [33]	infinite loop†
uniq utx 4.3	1146	1146	duplicate text processing	segmentation fault
look utx 4.3	1169	1169	dictionary lookup	segmentation fault
look svr 4.0 1.1	1363	1363	dictionary lookup	infinite loop
units svr 4.0 1.1	1504	1504	metric conversion	segmentation fault
deroff utx 4.3	2236	2236	document processing	segmentation fault
nullhttpd 0.5.0	5575	5575	webserver	remote heap buffer overflow (code)†
openldap 2.2.4	292598	6519	directory protocol	non-overflow denial of service†
ccrypt 1.2	7515	7515	encryption utility	segmentation fault†
indent 1.9.1	9906	9906	source code processing	infinite loop
lighttpd 1.4.17	51895	3829	webserver	remote heap buffer overflow (variables)†
flex 2.5.4a	18775	18775	lexical analyzer generator	segmentation fault
atris 1.0.6	21553	21553	graphical tetris game	local stack buffer exploit†
php 4.4.5	764489	5088	scripting language	integer overflow†
wu-ftpd 2.6.0	67029	35109	FTP server	format string vulnerability†
total	1246803	121337		

Fig. 5. Benchmark programs used in our experiments, with size of the program and the repaired program segment in lines of code. The Unix utilities are repaired in their entirety. However, for example, while the entire **wu-ftpd** server was processed as a unit, a smaller **io** module of **openldap** was selected for repair. A † indicates an openly available exploit.

$\mathcal{O}(n^2)$ worst case [32]. This minimized set of changes is the *final repair*. DIFFX edits can be converted automatically to standard **diff** patches, which can either be applied automatically to the system or presented to developers for inspection. In this paper, patch sizes are reported in the number of lines of a Unix **diff** patch, not DIFFX operations.

4 Repair Descriptions

In this section, we substantiate the claim that automated repair of real-world defects is possible by describing several buggy programs and examples of the patches that GenProg generates. The benchmarks for all experiments in this and subsequent sections are shown in Fig. 5. The defects considered include infinite loops, segmentation faults, several types of memory allocation errors, integer overflow, and a well-known format string vulnerability. In most cases, we consider all of the program source when making a repair; in a few cases we restrict attention to the single module visited by the negative test case. **gcd** is a small example based on Euclid's algorithm for computing greatest common divisors. **zune** is a fragment of code that caused all Microsoft Zune media players to freeze on 31 December 2008. The Unix utilities were taken from Miller et al.'s work on *fuzz testing*, in which programs crash when given random inputs [34]. The remaining benchmarks are taken from public vulnerability reports.

In the following sections, we describe several case studies of several exemplar repairs, only one of which has been previously published. The case studies are all taken from the security domain, but they illustrate the repair process in the context of large programs with publicly documented bugs. In each case, we first describe the bug that corresponds to a public vulnerability report; we then describe an indicative patch discovered by GenProg.

4.1 nullhttpd: Remote Heap Buffer Overflow

The **nullhttpd** webserver is a lightweight multithreaded webserver that handles static content as well as CGI scripts. Version 0.5.0 contains a heap-based buffer overflow vulnerability that allows remote attackers to execute arbitrary code (Section 2 illustrates this vulnerability for explanatory purposes). **nullhttpd** trusts the **Content-Length** value provided by the user in the HTTP header of POST requests; negative values cause **nullhttpd** to overflow a buffer.

We used six positive test cases that include both GET and POST requests and a publicly available exploit to create the negative test case. The negative test case request crashes the webserver, which is not set to respawn. To determine if the attack succeeded we insert a legitimate request for **index.html** after the exploit; the negative test case fails if the correct **index.html** is not produced.

The actual buffer overflow occurs in the **ReadPOST Data()** function, defined in **http.c**:

```
108  // http.c
109  conn[sid].PostData=
110      calloc(conn[sid].dat->in_ContentLength+1024,
111             sizeof(char));
112  pPostData=conn[sid].PostData;
113  ...
114  do {
115    rc=recv(conn[sid].socket,
116            pPostData, 1024, 0); /* overflow! */
117    ...
118    pPostData+=rc;
119  } while ((rc==1024) ||
120           (x<conn[sid].dat->in_ContentLength));
```

The value **in_ContentLength** is supplied by the attacker. However, there is a second location in the program, the **cgi_main()** function on line 267 of **cgi.c**, where POST-data are processed and copied:

```
267  // cgi.c
268  if (conn[sid].dat->in_ContentLength>0) {
269    write(local.out, conn[sid].PostData,
270          conn[sid].dat->in_ContentLength);
271  }
```

The evolved repair changes the high-level **read_header()** function so that it uses the POST-data processing in **cgi_main()** instead of calling **ReadPost Data**. The final, minimized repair is five lines long.

Although the repair is not the one supplied in the next release by human developers—which inserts local bounds-checking code in **ReadPOSTData()**—it both eliminates the vulnerability and retains desired functionality.

4.2 openldap: Nonoverflow Denial of Service

The **openldap** server implements the lightweight directory access protocol, allowing clients to authenticate and make queries (e.g., to a company's internal telephone directory). Version 2.3.41 is vulnerable to a denial of service attack. LDAP encodes protocol elements using a lightweight basic encoding rule (BER); nonauthenticated remote attackers can crash the server by making improperly formed requests.

The assertion visibly fails in **liblber/io.c**, so we restricted attention to that single file to demonstrate that we can repair program modules in isolation without requiring a whole-program analysis. To evaluate the fitness of a variant **io.c** we copied it in to the **openldap** source tree and ran **make** to rebuild and link the **liblber** library, then applied the test cases to the resulting binary.

The positive test cases consist of an unmodified 25-second prefix of the regression suite that ships with **openldap**. The negative test case was a copy of a positive test case with an exploit request inserted in the middle:

```
perl -e 'print"\xff\xff\xff\x00\x84\x41\x42\x43\x44"'
  | nc $HOST $PORT
```

The problematic code is around line 522 of `io.c`:

```
516 for (i=1; (char *)p<ber->ber_rwptr; i++) {
517   tag <<= 8;
518   tag |= *p++;
519   if (!(tag & LBER_MORE_TAG_MASK)) break;
520   if (i == sizeof(ber_tag_t)-1) {
521     /* Is the tag too big? */
522     sock_errset(ERANGE); /* !! buggy assert */
523     return LBER_DEFAULT;
524   }
525 }
526 if ((char *)p == ber->ber_rwptr) {
527   /* Did we run out of bytes? */
528   sock_errset(EWOULDBLOCK);
529   return LBER_DEFAULT;
530 }
```

The **for** loop contains both a sanity check and processing for large **ber** tags. The first 127 tag values are represented with a single byte: If the high bit is set, the next byte is used as well, and so on. The repair removes the entire loop (lines 516-524), leaving the "**run out of bytes**" check untouched. This limits the number of BER tags that the repaired **openldap** can handle to 127. A more natural repair would be to fix the sanity check while still supporting multibyte BER tags. However, only about 30 tags are actually defined for **openldap** requests, so the repair is fine for all **openldap** uses, and passes all the tests.

4.3 lighttpd: Remote Heap Buffer Overflow

lighttpd is a webserver optimized for high-performance environments; it is used by **YouTube** and **Wikimedia**, among others. In Version 1.4.17, the **fastcgi** module, which improves script performance, is vulnerable to a heap buffer overflow that allows remote attackers to overwrite arbitrary CGI variables (and thus control what is executed) on the server machine. In this case, GenProg repaired a

```
1  POST /hello.php HTTP/1.1
2  Host: localhost:8000
3  Connection: close
4  Content-length: 21213
5  Content-Type: application/x-www-form-urlencoded
6  ...
7  randomly-generated text
8  ...
9  \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
10 SCRIPT_FILENAME/etc/passwd
```

Fig. 6. Exploit POST request for **lighttpd**. The random text creates a request of the correct size; line 9 uses a fake FastCGI record to mark the end of the data. Line 10 overwrites the execute script so that the vulnerable server responds with the contents of **/etc/passwd**.

dynamically linked shared object, **mod_fastcgi.so**, without touching the main executable.

The positive test cases included requests for static content (i.e., **GET index.html**) and a request to a 50-line CGI Perl script which, among other actions, prints all server and CGI environment variables. The negative test case is the request shown in Fig. 6, which uses a known exploit to retrieve the contents of **/etc/passwd**—if the file contents are not returned, the test case passes.

The key problem is with the **fcgi_env_add** function, which uses **memcpy** to add data to a buffer without proper bounds checks. **fcgi_env_add** is called many times in a loop by **fcgi_create_env**, controlled by the following bounds calculation:

```
2051 off_t weWant =
2052   req_cq->bytes_in - offset > FCGI_MAX_LENGTH
2053   ? FCGI_MAX_LENGTH : req_cq->bytes_in - offset;
```

The repair modifies this calculation to:

```
2051 off_t weWant =
2052   req_cq->bytes_in - offset > FCGI_MAX_LENGTH
2053   ? FCGI_MAX_LENGTH : weWant;
```

weWant is thus uninitialized, causing the loop to exit early on very long data allocations. However, the repaired server can still report all CGI and server environment variables and serve both static and dynamic content.

4.4 php: Integer Overflow

The **php** program is an interpreter for a popular web-application scripting language. Version 5.2.1 is vulnerable to an integer overflow attack that allows context-dependent attackers to execute arbitrary code by exploiting the way the interpreter calculates and maintains bounds on string objects in single-character string replacements. As with the `openldap` repair example, we restricted GenProg's operations to the string processing library.

We manually generated three positive test cases that exercise basic PHP functionality, including iteration, string splitting and concatenation, and popular built-in functions such as **explode**. The negative test case included basic PHP string processing before and after the following exploit code:

```
str_replace("A", str_repeat("B", 65535),
                 str_repeat("A", 65538));
```

A program variant passed this test if it produced the correct output without crashing.

Single-character string replacement replaces every instance of a character ("**A**" in the attack) in a string (65,538 "**A**"s) with a larger string (65,535 "**B**"s). This functionality is implemented by **php_char_to_str_ex**, which is called

by function **php_str_replace_in_subject** at line 3478 of file **string.c**:

```
3476  if (Z_STRLEN_P(search) == 1) {
3477    php_char_to_str_ex(
3478     Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
3479     Z_STRVAL_P(search)[0], Z_STRVAL_P(replace),
3480     Z_STRLEN_P(replace), result,
3481     case_sensitivity, replace_count);
3482  } else if (Z_STRLEN_P(search) > 1) {
3483    Z_STRVAL_P(result) =
3484     php_str_to_str_ex(
3485      Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
3486      Z_STRVAL_P(search), Z_STRLEN_P(search),
3487      Z_STRVAL_P(replace), Z_STRLEN_P(replace),
3488      &Z_STRLEN_P(result), case_sensitivity,
3489      replace_count);
3490  } else {
3491    *result = **subject;
3492    zval_copy_ctor(result);
3493    INIT_PZVAL(result);
3494  }
```

php_str_replace_in_subject uses a macro **Z_STRLEN_P**, defined in a header file, to calculate the new string length. This macro expands to **len + (char_count * (to_len - 1))** on line 3,480, wrapping around to a small negative number on the exploitative input. The repair changes lines 3,476-3,482 to:

```
3476  if (Z_STRLEN_P(search) != 1) {
```

Single-character string replaces are thus disabled, with the output set to an unchanged copy of the input, while multicharacter string replaces, performed by **php_str_to_str_ex**, work as before. The **php_str_to_str_ex** function replaces every instance of one substring with another and is not vulnerable to the same type of integer overflow as **php_char_to_str_ex** because it calculates the resulting length differently. Disabling functionality to suppress a security violation is often a legitimate response in this context: Many systems can be operated in a "safe mode" or "read-only mode." Although acceptable in this situation, disabling functionality could have deleterious consequences in other settings; we address this issue in Section 6.2.

4.5 wu-ftpd: Format String

wu-ftpd is an FTP server that allows for anonymous and authenticated file transfers and command execution. Version 2.6.0 is vulnerable to a well-known format string vulnerability. If **SITE EXEC** is enabled, a user can execute a restricted subset of quoted commands on the server. Because the user's command string is passed directly to a **printf**-like function, anonymous remote users gain shell access by using carefully selected conversion characters. Although the exploit is similar in structure to a buffer overrun, the underlying problem is a lack of input validation. GenProg operated on the entire **wu-ftpd** source.

We used five positive test cases (obtaining a directory listing, transferring a text file, transferring a binary file, correctly rejecting an invalid login, and an innocent **SITE EXEC** command). The negative test used an posted exploit to dynamically craft a format string for the target architecture.

The bug is in the **site_exec()** function of **ftpcmd.y**, which manipulates the user-supplied buffer **cmd**:

```
1875  /* sanitize the command-string */
1876  if (sp == 0) {
1877    while ((slash = strchr(cmd, '/')) != 0)
1878      cmd = slash + 1;
1879  } else {
1880    while (sp
1881           && (slash = (char *) strchr(cmd, '/'))
1882           && (slash < sp))
1883      cmd = slash + 1;
1884    for (t = cmd; *t && !isspace(*t); t++) {
1885      if (isupper(*t)) *t = tolower(*t);
1886    }
1887  }
1888  ...
1889  lreply(200, cmd);
1890  /* !!! vulnerable lreply call */
1891  ...
1892  /* output result of SITE EXEC */
1893  lreply(200, buf);
```

lreply(x,y,z...) provides logging output by printing the executing command and providing the return code (**200** denotes success in the FTP protocol). The **lreply(200,cmd)** on line 1,889 calls **printf(cmd)**, which, with a carefully crafted **cmd** format string, compromises the system. The explicit attempt to sanitize **cmd** by skipping past slashes and converting to lowercase does not prevent format-string attacks. The repair replaces **lreply(200,cmd)** with **lreply(200, (char *)"")**, which disables verbose debugging output on **cmd** itself, but does report the return code and the properly sanitized **site_exec** in **buf** while maintaining required functionality.

5 GenProg Repair Performance

This section reports the results of experiments that use GenProg to repair errors in multiple legacy programs: 1) evaluating repair success over multiple trials and 2) measuring performance and scalability in terms of fitness function evaluations and wall-clock time.

5.1 Experimental Setup

Programs and Defects. The benchmarks consist of all programs in Fig. 5. These programs total 1.25M LOC; the repaired errors span eight defect classes (infinite loop, segmentation fault, remote heap buffer overflow to inject code, remote heap buffer overflow to overwrite variables, nonoverflow denial of service, local stack buffer overflow, integer overflow, and format string vulnerability) and are repaired in 120K lines of module or program code. Our experiments were conducted on a quad-core 3 GHz machine.

Test cases. For each program, we used a single negative test case that elicits the given fault. For the Unix utilities, we selected the first fuzz input that evinced a fault; for the others, we constructed test cases based on the vulnerability reports (see Section 4, for examples). We selected a small number (e.g., 2-6) of positive test cases per program. In some cases, we used noncrashing fuzz inputs; in others, we manually created simple cases, focusing on testing relevant program functionality; for **openldap**, we used part of its test suite.

Parameters. We report results for one set of global GenProg parameters that seemed to work well. We chose $\text{pop\_size} = 40$, which is small compared to typical GP applications; on each trial, we ran the GP for a maximum of 10 generations (also a small number). For fitness

computation, we set $W_{PosT} = 1$ and $W_{NegT} = 10$. In related work [35], we note that it is possible to select more precise weights, as measured by the fitness distance correlation metric [36]. However, we find that the values used here work well on our benchmark set. These heuristically chosen values capture our intuition that the fitness function should emphasize repairing the fault and that the positive test cases should be weighted evenly. We leave a more thorough exploration for future work.

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 statements executed by both the negative test case and any positive test case will not be mutated, and $W_{Path} = 0.01$ means such statements will be considered infrequently. The parameter set $W_{Path} = 0.01$ and $W_{mut} = 0.06$ works well in practice. Additional experiments show that GenProg is robust to changes in many of these parameters, such as population size, and that varying the selection or crossover techniques has a small impact on time to repair or success [26]. We have experimented with higher probabilities, finding that success worsens beyond $W_{mut} > 0.12$.

The *weighted path length* is the weighted sum of statements on the negative path and provides one estimate of the complexity of the search space. Statements that appear only on the negative path receive a weight of 1.0, while those also on a positive path receive a weight of W_{Path}. This metric is correlated with algorithm performance (Section 5.3).

Trial. We define a *trial* to consist of at most two serial invocations of the GP loop using the parameter sets above in order. We stop a trial if an initial repair is found; otherwise, the GP is run for 10 generations per parameter set. We performed 100 random trials for each program and report the percentage of trials that produce a repair, average time to the initial repair in a successful trial, and time to minimize a final repair, a deterministic process performed once per successful trial.

An initial repair is one that passes all input test cases. Given the same random seed, each trial is deterministically reproducible and leads to the same repair. With unique seeds and for some programs, GenProg generates several different patches over many random trials. For example, over 100 random trials, GenProg produces several different acceptable patches for **ccrypt**, but only ever produces one such patch for **openldap**. Such disparities are likely related to the program, error, and patch type. We do not report the number of different patches found because, in theory, there are an infinite number of ways to address any particular error. However, we note that our definition of repair as a set of changes that cause a program to pass all test cases renders all such patches "acceptable." Ranking of different but acceptable patches remains an area of future investigation.

Optimizations. When calculating fitness, we memorize fitness results based on the pretty-printed abstract syntax tree so that two variants with different ASTs but identical source code are not evaluated twice. Similarly, variants that are copied unchanged to the next generation are not reevaluated. Beyond such caching, the prototype tool is not optimized. In particular, we do not take advantage of the fact that the GP repair task is embarrassingly parallel: Both the fitness of all variant programs and also the test cases for any individual variant can all be evaluated independently [25].

5.2 Repair Results

Fig. 7 summarizes repair results for 16 C programs. The "Initial Repair" heading reports timing information for the GP phase and does not include the time for repair minimization. The "Time" column reports the average wall-clock time per trial that produced a primary repair; execution time is analyzed in more detail in Section 5.3. Repairs are found in 357 seconds on average. The "fitness" column shows the average number of fitness evaluations per successful trial, which we include because fitness function evaluation is the dominant expense in most GP applications and the measure is independent of specific hardware configuration. The "Success" column gives the fraction of trials that were successful. On average, over 77 percent 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. The "Size" column lists the size of the primary repair **diff** in lines.

The "Final Repair" heading gives performance information for transforming the primary repair into the final repair and a summary of the effect of the final repair, as judged by manual inspection. Minimization is deterministic and takes less time and fewer fitness evaluations than the initial repair process. The final minimized patch is quite manageable, averaging 5.1 lines.

Of the 16 patches, seven insert code (**gcd**, **zune**, **look-u**, **look-s**, **units**, **ccrypt**, and **indent**), seven delete code (**uniq**, **deroff**, **openldap**, **lighttpd**, **flex**, **atris**, and **php**), and two both insert and delete code (**nullhttpd** and **wu-ftpd**). Note that this does not speak to the sequence of mutations that lead to a given repair, only the operations in the final patch: A swap followed by a deletion may result in a minimized patch that contains only an insertion.

While a comprehensive code review is beyond the scope of this paper, manual inspection suggests that the produced patches are acceptable. We note that patches that delete code do not necessarily degrade functionality: The deleted code may have been included erroneously, or the patch may compensate for the deletion with an insertion. The **uniq**, **deroff**, and **flex** patches delete erroneous code and do not degrade untested functionality. The **openldap** patch removes unnecessary faulty code (handling of multibyte BER tags, when only 30 tags are used), and thus does not degrade functionality in practice. The **nullhttpd** and **wu-ftpd** patches delete faulty code and replace them by inserting nonfaulty code found elsewhere. The **wu-ftpd** patch disables verbose logging output in one source location, but does not modify the functionality of the program itself, and the **nullhttpd** patch does not degrade functionality. The effect of the **lighttpd** patch is machine-specific: It may reduce functionality on very long messages, though, in our experiments, it did not. More detailed patch

Program	Positive Tests	|Path|	Initial Repair Time	Fitness	Success	Size	Final Repair Time	Fitness	Size	Effect
gcd	5x human	1.3	153 s	45.0	54%	21	4 s	4	2	Insert
zune	6x human	2.9	42 s	203.5	72%	11	1 s	2	3	Insert
uniq	5x fuzz	81.5	34 s	15.5	100%	24	2 s	6	4	Delete
look-u	5x fuzz	213.0	45 s	20.1	99%	24	3 s	10	11	Insert
look-s	5x fuzz	32.4	55 s	13.5	100%	21	4 s	5	3	Insert
units	5x human	2159.7	109 s	61.7	7%	23	2 s	6	4	Insert
deroff	5x fuzz	251.4	131 s	28.6	97%	61	2 s	7	3	Delete
nullhttpd	6x human	768.5	578 s	95.1	36%	71	76 s	16	5	Both
openldap	40x human	25.4	665 s	10.6	100%	73	549 s	10	16	Delete
ccrypt	6x human	18.01	330 s	32.3	100%	34	13 s	10	14	Insert
indent	5x fuzz	1435.9	546 s	108.6	7%	221	13 s	13	2	Insert
lighttpd	3x human	135.8	394 s	28.8	100%	214	139 s	14	3	Delete
flex	5x fuzz	3836.6	230 s	39.4	5%	52	7 s	6	3	Delete
atris	2x human	34.0	80 s	20.2	82%	19	11 s	7	3	Delete
php	3x human	30.9	56 s	15.5	100%	139	94 s	11	10	Delete
wu-ftpd	5x human	149.0	2256 s	48.5	75%	64	300 s	6	5	Both
average		573.52	356.5 s	33.63	77.0%	67.0	76.3 s	8.23	5.7	

Fig. 7. Experimental results on 120K lines of program or module source code from programs totaling 1.25M lines of source code. We report averages for 100 random trials. The "Positive Tests" column describes the positive tests. The "|Path|" columns give the weighted path length. "Initial Repair" gives the average performance for one trial, in terms of "Time" (the average time taken for each successful trial), "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 Unix **diff** size between the original source and the primary repair, in lines. "Final Repair" reports the same information for the production of a 1-minimal repair from the first initial repair found; the minimization process always succeeds. "Effect" describes the operations performed by an indicative final patch: A patch may insert code, delete code, or both insert, and delete code.

descriptions are provided in Section 4, above; we evaluate repair quality using indicative workloads and held-out fuzz testing in Section 6.

In many cases it is also possible to insert code without negatively affecting the functionality of a benchmark program. The **zune** and **gcd** benchmarks both contain infinite loops: **zune** when calculating dates involving leap years, and **gcd** if one argument is zero. In both cases, the repair involves inserting additional code: For **gcd**, the repair inserts code that returns early (skipping the infinite loop) if the argument is zero. In **zune**, code is added to one of three branches that decrements the day in the main body of the loop (allowing leap years with exactly 366 days remaining to be processed correctly). In both of these cases, the insertions are carefully guarded so as to apply only to relevant inputs (i.e., zero-valued arguments or tricky leap years), which explains why the inserted code does not negatively impact other functionality. Similar behavior is seen for **look-s**, where a buggy binary search over a dictionary never terminates if the input dictionary is not presorted. Our repair inserts a new exit condition to the loop (i.e., a guarded **break**). A more complicated example is **units**, in which user input is read into a static buffer without bounds checks, a pointer to the result is passed to a **lookup()** function, and the result of **lookup()** is possibly dereferenced. Our repair inserts code into **lookup()** so that it calls an existing initialization function on failure (i.e., before the **return**), reinitializing the static buffer and avoiding the segfault. Combined with the explanations of repairs for **nullhttpd** (Section 4.1) and **wuftpd** (Section 4.5), which include both insertions and deletions, these changes are indicative of repairs involving inserted code.

This experiment demonstrates that GenProg can successfully repair a number of defect types in existing programs in a reasonable amount of time. Reports suggest that it takes human developers 28 days on average to address even security-critical repairs [37]; nine days elapsed between the posted exploit source for **wu-ftpd**, and the availability of its patch.

5.3 Scalability and Performance

GenProg is largely CPU-bound. An average repair run took 356.5 seconds. Fig. 8 shows the proportion of time taken by each important component. Executing the test cases for the fitness function takes much of this time: on average, positive test cases take $29.76\% \pm 24.0$ and negative test cases $32.99\% \pm 23.17$ of the time. In total, fitness evaluations comprise $62.75\% \pm 30.37$ of total repair time. Many test cases include time outs (e.g., negative test cases that specify an infinite-loop error); others involve explicit internal delays (e.g., ad hoc instructions to wait 2 seconds for the web server to get "up and running" before requests are sent; the **openldap** test suite makes extensive use of this type of delay), contributing to their runtime. Compilation of variants averaged $27.13\% \pm 22.55$ of repair time. Our initial implementation makes no attempt at incremental compilation. The high standard deviations arise from the widely varying test suite execution times (e.g., from 0.2 seconds for **zune** to 62.7 seconds for **openldap**).

Fig. 9 plots weighted path length against search time, measured as the average number of fitness evaluations until the first repair, on a log-log scale. The straight line suggests

Program	Fitness Total	Positive Tests	Negative Tests	Compile
`gcd`	91.2%	44.4%	46.8%	8.4%
`zune`	94.7%	23.2%	71.5%	3.7%
`uniq`	71.0%	17.2%	53.8%	25%
`look-u`	76.4%	17.1%	59.3%	20.7%
`look-s`	83.8%	29.9%	53.9%	12.9%
`units`	57.8%	20.7%	37.1%	35.5%
`deroff`	44.5%	8.9%	35.6%	46.5%
`nullhttpd`	74.9%	22.9%	52.0%	12.8%
`openldap`	93.7%	82.6%	11.1%	6.2%
`indent`	36.5%	16.4%	20.1%	43.1%
`ccrypt`	87.3%	65.2%	22.0%	4.7%
`lighttpd`	68.0%	67.9%	0.06%	25.3%
`flex`	23.2%	18.3%	4.8%	39.5%
`atris`	1.9%	0.8%	1.1%	64.9%
`php`	12.0%	2.0%	10.0%	78.4%
`wu-ftpd`	87.2%	38.6%	48.6%	6.6%
Average	62.75	29.76	32.99	27.14
StdDev	30.37	24.00	23.17	22.55

Fig. 8. Percentage of total repair time spent on particular repair tasks.

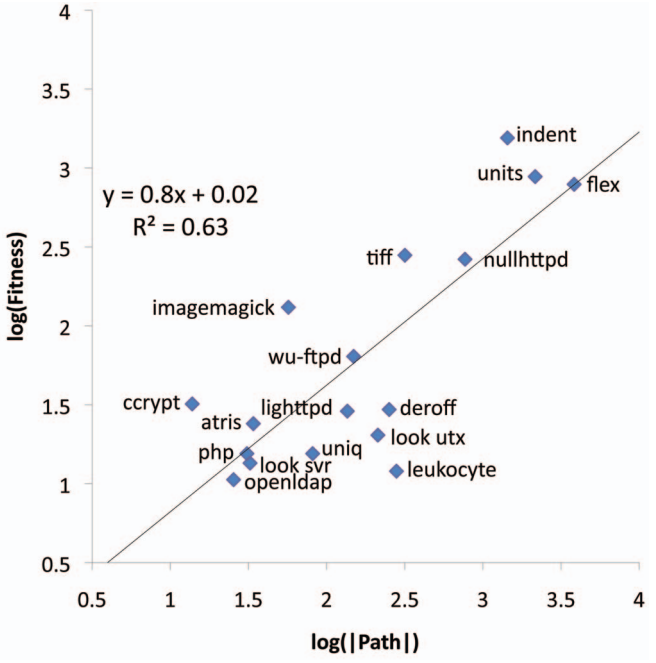

Fig. 9. GenProg execution time scales with weighted path size. Data are shown for 17 benchmark programs, including some not described here (included for increased statistical significance; see [35] for details on the additional benchmarks) and excluding `gcd` and `zune`. The x-axis shows weighted path length; the y-axis shows the number of fitness evaluations performed before a primary repair is found (averaged over 100 runs). Note the base-10 log-log scale.

a relationship following a power law of the form $y = ax^b$, where b is the best-fit slope and $b = 1$ indicates a linear relationship. Fig. 9 suggests that the relationship between path length and search time is less than linear with slope 0.8. Recall that the weighted path is based on observed test case behavior and not on the much larger number of loop-free paths in the program. We note that weighted path length does not fully measure the complexity of the search space; notably, as program size grows, the number of possible statements that could be swapped or inserted along the path grows, which is not accounted for in the weighted path length. Accordingly, this relationship is only an approximation of scalability, and search time may not grow sublinearly with search space using other measures. However, the results in Fig. 9 are encouraging, because they suggest that search time is governed more by weighted path rather than program size.

The test cases comprise fitness evaluation and define patch correctness; test suite selection is thus important to both scalability and correctness. For example, when repairing `nullhttpd` without a positive test case for POST-data functionality, GenProg generates a repair that disables POST functionality entirely. In this instance, all of the POST-processing functionality is on the weighted path (i.e., visited by the negative test case but not by any positive test cases) and deleting those statements is the most expedient way to find a variant that passes all tests. As a quick fix this is not unreasonable and is safer than the common alarm practice of running in read-only mode. However, including the POST-functionality test case leads GenProg to find a repair that does not remove functionality. Adding positive test cases can actually reduce the weighted path length while protecting core functionality, and thus improve the success rate while possibly also increasing runtime. Experiments have shown that larger test suites increase fitness variability in early GP generations [26]; additional experiments confirm that test suite selection techniques can improve the performance of GenProg on programs with large regression suites, reducing repair times by up to 80 percent while finding the same repairs [35].

These results suggest that GenProg can repair off-the-shelf code in a reasonable amount of time, that GenProg performance scales with the size of the weighted path, and that there are several viable avenues for applying the technique to larger programs with more comprehensive test suites in the future.

6 GenProg Repair Quality

Although the results of the previous sections are encouraging, they do not systematically address the important issue of repair quality. GenProg's reliance on positive test cases provides an important check against lost functionality. The use of test cases exclusively to define acceptability admits the possibility of repairs that degrade the quality of the design of a system or make a system more difficult to maintain, concerns that are difficult to evaluate automatically and are beyond the scope of this paper. However, certain dangers posed by, for example, inadequate test suites—such as repairs that reduce functionality or introduce vulnerabilites—can be evaluated automatically using indicative workloads, held-out test cases, and fuzz testing.

Additionally, the claim of *automated* program repair relies on manual initialization and dispatch of GenProg. In principle, automated detection techniques could signal the repair process to complete the automation loop. Integrating GenProg with automated error detection produces a closed-loop error detection and repair system that would allow us

Case	IDS Class.	Repair Made?	Behavior In This Case
1	True Neg	—	Common case: legitimate request handled correctly.
2	False Neg	—	Attack succeeds. Repair not attempted.
3	True Pos	Yes	Attack stopped, bug fixed. Poor repair may drop requests.
4	True Pos	No	Attack detected, but bug not repaired.
5	False Pos	No	Valid request dropped. No repair found.
6	False Pos	Yes	Valid request dropped. Poor "repair" may drop requests.

Fig. 10. Closed-loop system outcomes (per request), as a function of anomaly detector and repair success. Cases 3 and 6 are new concerns.

to study repair quality and overhead on programs with realistic workloads.

This section therefore evaluates GenProg in the context of a proof-of-concept closed-loop system for webserver-based programs, with several experimental goals:

1. outline the prototype closed-loop repair system and enumerate new experimental concerns,
2. measure the performance impact of repair time and quality on a real, running system, including the effects of a functionality reducing repair on system throughput,
3. analyze the quality of the generated repairs in terms of functionality using fuzz testing and variant bug-inducing input, and
4. measure the costs associated with intrusion-detection system (IDS) false positives.

6.1 Closed-Loop System Overview

Our proposed closed-loop repair system has two requirements beyond the input required by GenProg: 1) anomaly detection in near-real time, to provide a signal to launch the repair process, and 2) the ability to record and replay system input [38] so we can automatically construct a negative test case. Anomaly detection could be provided by existing behavior-based techniques that run concurrently with the program of interest, operating at almost any level (e.g., by monitoring program behavior, examining network traffic, using saved state from regular checkpoints, etc.). Our prototype adopts an intrusion-detection system that detects suspicious HTTP requests based on request features [39]. In a preprocessing phase, the IDS learns a probabilistic finite state machine model of normal requests using a large training set of legitimate traffic. After training, the model labels subsequent HTTP requests with a probability corresponding to "suspiciousness."

Given these components, the system works as follows: While the webserver is run normally and exposed to untrusted inputs from the outside world, the IDS checks for anomalous behavior, and the system stores program state and each input while it is being processed. When the IDS detects an anomaly, the program is suspended, and GenProg is invoked to repair the suspicious behavior. The negative test case is constructed from the IDS-flagged input: A variant is run in a sandbox on the input with the program state stored from just before the input was detected. If the variant terminates successfully without triggering the IDS, the negative test case passes; otherwise, it fails. The positive tests consist of standard system regression tests. For the purpose of these experiments, we use the tests described in Section 5 to guide the repair search, and add new, large indicative workloads to evaluate the effect of the repair search and deployment on several benchmarks.

If a patch is generated, it can be deployed immediately. If GenProg cannot locate a viable repair within the time limit, subsequent identical requests should be dropped and an operator alerted. While GenProg runs, the system can either refuse requests, respond to them in a "safe mode" [40], or use any other technique (e.g., fast signature generation [41]) to filter suspicious requests. Certain application domains (e.g., supply chain management requests, banking, or e-commerce) support buffering of requests received during the repair procedure, so they can be processed later.

Fig. 10 summarizes the effects of the proposed system on a running program; these effects depend on the anomaly detector's misclassification rates (false positives/negatives) and the efficacy of the repair method. The integration of GenProg with an IDS creates two new areas of particular concern. The first new concern, Case 3, is the effect of an imperfect repair (e.g., one that degrades functionality not guaranteed by the positive tests) to a true vulnerability, which can potentially lead to the loss of legitimate requests or, in the worst case, new vulnerabilities. For security vulnerabilities in particular, any repair system should include a strong final check of patch validity before deployment. To evaluate the suitability of GenProg on real systems, it is therefore important to gain confidence, first that GenProg repairs underlying errors and second that it is unlikely to introduce new faults. In Case 6, a "repair" generated in response to an IDS false alarm could also degrade functionality, again losing legitimate requests.

The remainder of this section evaluates these concerns, and uses them as a framework to motivate and guide the evaluation of automated repair quality and overhead, in terms of their effect on program throughput and correctness, measured by held-out test suites and indicative workloads.

6.2 Experimental Setup

We focus the repair quality experiments on three of our benchmarks that consist of security vulnerabilities in long-running servers: **`lighttpd`**, **`nullhttpd`**, and **`php`**. There exist many mature intrusion-detection systems for security vulnerabilities, providing a natural means of identifying bugs to be repaired. Similarly, web servers are a compelling starting point for closed-loop repair: They are common attack targets, they are important services that run continually, and they are event driven, making it easier to isolate negative test cases. Note that for the **`php`** experiments we repair the **`php`** interpreter used by an unchanging, off-the-shelf **`apache`** webserver, in **`libphp.so`**.

Program	Repair Made?	Requests Lost to Repair Time	Requests Lost to Repair Quality	Fuzz Test Failures			
				Generic		Exploit	
				Before	After	Before	After
`nullhttpd`	Yes	$2.38\% \pm 0.83\%$	$0.00\% \pm 0.25\%$	0	0	10	0
`lighttpd`	Yes	$2.03\% \pm 0.37\%$	$0.03\% \pm 1.53\%$	1410	1410	9	0
`php`	Yes	$0.12\% \pm 0.00\%$	$0.02\% \pm 0.02\%$	3	3	5	0
Quasi False Pos. 1	Yes	$7.83\% \pm 0.49\%$	$0.00\% \pm 2.22\%$	0	0	—	
Quasi False Pos. 2	Yes	$3.04\% \pm 0.29\%$	$0.57\% \pm 3.91\%$	0	0	—	
Quasi False Pos. 3	No	$6.92\% \pm 0.09\%$	—	—			

Fig. 11. Closed-loop repair system evaluation. Each row represents a different repair scenario and is separately normalized so that the prerepair daily throughput is 100 percent. The `nullhttpd` and `lighttpd` rows show results for true repairs. The `php` row shows the results for a repair that degrades functionality. The False Pos. rows show the effects of repairing three intrusion detection system false positives on `nullhttpd`. The number after $\pm$ indicates one standard deviation. "Lost to Repair Time" indicates the fraction of the daily workload lost while the server was offline generating the repair. "Lost to Repair Quality" indicates the fraction of the daily workload lost after the repair was deployed. "Generic Fuzz Test Failures" counts the number of held-out fuzz inputs failed before and after the repair. "Exploit Failures" measures the held-out fuzz exploit tests failed before and after the repair.

Several experiments in this section use indicative workloads to measure program throughput pre, during, and postrepair. We obtained workloads and content layouts from the University of Virginia CS Department webserver. To evaluate repairs to the **`nullhttpd`** and **`lighttpd`** webservers, we used a workload of 138,226 HTTP requests spanning 12,743 distinct client IP addresses over a 14-hour period on 11 November 2008. To evaluate repairs to **`php`**, we obtained the room and resource reservation system used by the University of Virginia CS Department, which features authentication, graphical animated date and time selection, and a **`mysql`** back end. It totals 16,417 lines of PHP, including 28 uses of **`str_replace`** (the subject of the **`php`** repair), and is a fairly indicative three-tier web application. We also obtained 12,375 requests to this system, spanning all of 11 November 2008. Recall that the `php` repair loses functionality; we use this workload to evaluate the effect of such a repair. In all cases, a request was labeled "successful" if the correct (bit-for-bit) data were returned to the client before that client started a new request; success requires both correct output and response time.

Our test machine contains 2 GB of RAM and a 2.4 GHz dual-core CPU. To avoid masking repair cost, we uniformly sped up the workloads until the server machine was at 100 percent utilization (and additional speedups resulted in dropped packets). To remove network latency and bandwidth considerations, we ran servers and clients on the same machine.

We use two metrics to evaluate repair overhead and quality. The first metric is the number of successful requests a program processed before, during, and after a repair. To evaluate repair time overhead, we assume a worst-case scenario in which the same machine is used both for serving requests and repairing the program and in which all incoming requests are dropped (i.e., not buffered) during the repair process. The second metric evaluates a program on held-out fuzz testing; comparing behavior pre and postrepair can suggest whether a repair has introduced new errors, and whether the repair generalizes.

6.3 The Cost of Repair Time

We first measure the overhead of running GenProg itself by measuring the number of requests from the indicative workloads the unmodified programs successfully handle. Next, we generated the repair, noting the requests lost during the time taken to repair on the server machine. Fig. 11 summarizes the results. The "Requests Lost To Repair Time" column shows the requests dropped during the repair as a fraction of the total number of successful requests served by the original program. To avoid skewing relative performance by the size of the workload, the numbers have been normalized to represent a single day containing a single attack. Note that the absolute speed of the server is not relevant here: A server machine that was twice as fast overall would generate the repair in half the time, but would also process requests twice as quickly. Fewer than 8 percent of daily requests were lost while the system was offline for repairs. Buffering requests, repairing on a separate machine, or using techniques such as signature generation could reduce this overhead.

6.4 Cost of a Repair that Degrades Functionality

The "Requests Lost to Repair Quality" column of Fig. 11 quantifies the effect of the generated repairs on program throughput. This row shows the difference in the number of requests that each benchmark could handle before and after the repair, expressed as a percentage of total daily throughput. The repairs for **`nullhttpd`** and **`lighttpd`** do not noticeably affect their performance. Recall that the **`php`** repair degrades functionality by disabling portions of the **`str_replace`** function. The **`php`** row of Fig. 11 shows that this low quality (loss of functionality) repair does not strongly affect system performance. Given the low-quality repair's potential for harm, the low "Lost" percentage for **`php`** is worth examining. Of the reservation application's 28 uses of **`str_replace`**, 11 involve replacements of multicharacter substrings, such as replacing "`-`" with "`- -`" for strings placed in HTML comments. Our repair leaves multicharacter substring behavior unchanged. Many of the other uses of **`str_replace`** occur on rare paths. For example, in

```
$result = mysql_query($query) or
  die("Query failed : " . mysql_error());
while($data=mysql_fetch_array($result,ASSOC))
  if (!$element_label)
    $label = str_replace('_',' ',$data['Fld']);
  else
    $label = $element_label;
```

str_replace is used to make a form label, but is only invoked if another variable, **element_label**, is **null**. Other uses replace, for example, underscores with spaces in a form label field. Since the repair causes single-character **str_replace** to perform no replacements, if there are no underscores in the field, then the result remains correct. Finally, a few of the remaining uses were for SQL sanitization, such as replacing "," with "','". However, the application also uses **mysql_real_escape_string**, so it remains safe from such attacks.

6.5 Repair Generality and Fuzzing

The experiments in the previous sections suggest that GenProg repairs do not impair legitimate requests, an important component of repair quality. Two additional concerns remain. First, repairs must not introduce new flaws or vulnerabilities, even when such behavior is not tested by the input test cases. To this end, Microsoft requires that security-critical changes be subject to 100,000 fuzz inputs [42] (i.e., randomly generated structured input strings). Similarly, we used the SPIKE black-box fuzzer from immunitysec.com to generate 100,000 held-out fuzz requests using its built-in handling of the HTTP protocol. The "Generic" column in Fig. 11 shows the results of supplying these requests to each program. Each program failed no additional tests postrepair: For example, **lighttpd** failed the same 1,410 fuzz tests before and after the repair. Second, a repair must do more than merely memorize and reject the exact attack input: It must address the underlying vulnerability. To evaluate whether the repairs generalize, we used the fuzzer to generate 10 held-out variants of each exploit input. The "Exploit" column shows the results. For example, **lighttpd** was vulnerable to nine of the variant exploits (plus the original exploit attack), while the repaired version defeated all of them (including the original). In no case did GenProg's repairs introduce any errors that were detected by 100,000 fuzz tests, and in every case GenProg's repairs defeated variant attacks based on the same exploit, showing that the repairs were not simply fragile memorizations of the input.

The issue of repair generality extends beyond the security examples shown here. Note that because this particular experiment only dealt with the repair of security defects, fuzz testing was more applicable than it would be in the general case. Establishing that a repair to a generic software engineering error did not introduce new failures or otherwise "overfit" could also be accomplished with held-out test cases or cross validation.

6.6 Cost of Intrusion Detection False Positives

Finally, we examine the effect of IDS false positives when used as a signal to GenProg. We trained the IDS on 534,109 requests from an independent data set [39]; this process took 528 seconds on a machine with quad-core 2.8 GHz and 8 GB of RAM. The resulting system assigns a score to each incoming request ranging from 0.0 (anomalous) to 1.0 (normal). However, the IDS perfectly discriminated between benign and exploitative requests in the testing workloads (no false negatives or false positives), with a threshold of 0.02. Therefore, to perform these experiments, we randomly selected three of the lowest scoring normal requests (closest to being incorrectly labeled anomalous) and attempted to "repair" **nullhttpd** against them, using the associated requests as input and a **diff** against the baseline result for the negative test case; we call these requests quasi-false positives (QFPs). The "False Pos." rows of Fig. 11 show the effect of time to repair and requests lost to repair when repairing these QFPs.

QFP #1 is a malformed HTTP request that includes quoted data before the **GET**:

```
"[11/Nov/2008:12:00:53 -0500]"
GET /people/modify/update.php HTTP/1.1
```

The GenProg repair changed the error response behavior so that the response header confusingly includes **HTTP/1.0** 200 OK while the user-visible body retains the correct **501 Not Implemented** message, but with the color-coding stripped. The header inclusion is ignored by most clients; the second change affects the user-visible error message. Neither causes the webserver to drop additional legitimate requests, and Fig. 11 shows no significant loss due to repair quality.

QFP #2 is a **HEAD** request; such requests are rarer than **GET** requests and only return header information such as last modification time. They are used by clients to determine if a cached local copy suffices:

```
HEAD /~user/mission_power_trace_movie.avi HTTP/1.0
```

The repair changes the processing of HEAD requests so that the **Cache-Control: no-store** line is omitted from the response. The **no-store** directive instructs the browser to store a response only as long as it is necessary to display it. The repair thus allows clients to cache pages longer than might be desired. It is worth noting that the **Expires: <date>** also included in the response header remains unchanged and correctly set to the same value as the **Date : <date>** header (also indicating that the page should not be cached), so a conforming browser is unlikely to behave differently. Fig. 11 indicates negligible loss from repair quality.

QFP #3 is a relatively standard HTTP request:

```
GET /~lcc-win32/lccwin32.css HTTP/1.1
```

GenProg fails to generate a repair within one run (240 seconds) because it cannot generate a variant that is successful at **GET index.html** (one of the positive test cases) but fails the almost identical QFP #3 request. Since no repair is deployed, there is no subsequent loss to repair quality.

These experiments support the claim that GenProg produces repairs that address the given errors and do not compromise functionality. It appears that the time taken to generate these repairs is reasonable and does not unduly influence real-world program performance. Finally, the experiments suggest that the danger from anomaly detection false positives is lower than that of low-quality repairs from inadequate test suites, but that both limitations are manageable.

7 Discussion, Limitations, and Threats

The experiments in Sections 5 and 6 suggest that GenProg can repair several classes of errors in off-the-shelf C programs efficiently. The experiments indicate that the overhead of GenProg is low, the costs associated with false

positives and low-quality repairs are low, that the repairs generalize without introducing new vulnerabilities, and that the approach may be viable when applied to real programs with real workloads, even when considering the additional concerns presented by a closed-loop detection and repair system. However, there are several limitations of the current work.

Nondeterminism. GenProg relies on test cases to encode both an error to repair and important functionality. Some properties are difficult or impossible to encode using test cases, such as nondeterministic properties; GenProg cannot currently repair race conditions, for example. We note, however, that many multithreaded programs, such as `nullhttpd`, can already be repaired if the threads are independent. This limitation could be mitigated by running each test case multiple times, incorporating scheduler constraints into the GP representation, and allowing a repair to contain both code changes and scheduling directives, or making multithreaded errors deterministic [43]. There are certain other classes of properties, such as liveness, fairness, and noninterference, that cannot be disproved with a finite number of execution examples; it is not clear how to test or patch noninterference information flow properties using our system.

Test suites and repair quality. GenProg defines repair acceptability according to whether the patched program passes the input test suite. Consequently, the size and scope of the test suite can directly impact the quality of the produced patch, even when minimized to reduce unnecessary changes. Because the test cases don't encode holistic design choices, the repairs produced by GenProg are not always the same as those produced by human developers. Repeated automatic patching could potentially degrade source code readability because, even though our patches are small in practice, they sometimes differ from those provided by human developers. Related research in automatic change documentation may mitigate this concern [44]. Repairs may reduce functionality if too few test cases are used, and the utility of the closed-loop architecture in particular requires that a test suite be sufficient to guard against lost functionality or new vulnerabilities. However, test cases are more readily available in practice than specifications or code annotations, and existing test case generation techniques [45] could be used to provide new positive or negative test cases and a more robust final check for patch validity. We found in Section 6 that several security-critical patches are robust in the face of fuzzed exploit inputs and do not appear to degrade functionality. Additionally, the experiments in Section 6 suggest that even repairs that reduce functionality do not produce prohibitive effects in practice; these results corroborate the precedent in previous work for this definition of repair "acceptability" [14]. Ultimately, however, GenProg is not designed to replace the human developer in the debugging pipeline, as it is unable, in its current incarnation, to consider higher level design goals or, in fact, any program behavior beyond that observed on test cases.

Results in Section 5.3 show that GenProg running time is dominated by fitness evaluations. Too many test cases may thus impede running time. However, GenProg has been shown to integrate well with test suite selection techniques [35], permitting speedups of 80 percent while finding the same repairs.

Fault localization. Fault localization is critical to the success of GenProg; without weighting by fault localization, our algorithm rarely succeeds (e.g., `gcd` fails 100 percent of the time). GenProg scalability is predicated on accurate fault localization using positive and negative test cases. In the current implementation, which makes use of a simple fault localization technique, GenProg scales well when the positive and negative test cases visit different portions of the program. In the case of security-related data-only attacks, where good and bad paths may overlap completely, the weighted path will not constrain the search space as effectively, potentially preventing timely repairs. More precise bug localization techniques [1] might mitigate this problem, though fault localization in general remains a difficult and unsolved problem. A related concern is GenProg's assumption that a repair can be adapted from elsewhere in the same source code. This limitation could potentially be addressed with a small library of repair templates to augment the search space. In the case of a very large code base, the randomized search process could be overwhelmed by too many statements to select from. In such cases, new methods could be developed for "fix localization." We leave further repair localization techniques as an avenue of future work.

Intrusion detection. For the closed-loop system described in Section 6.1, we used an intrusion detection system that does not apply to all fault types and does not actually locate the fault. We note that fault isolation by the IDS is not necessary to integrate with our proposed architecture because GenProg does its own fault isolation using existing techniques. Although the success of our approach is limited to faults that can be well-localized by lightweight techniques (e.g., excluding data-only attacks), it also means that we do not need to rely on an IDS that can pinpoint fault locations. Instead, our proposed closed-loop system requires only a monitoring system that can identify an input that leads to faulty behavior—a significantly easier problem—and that permits the construction of a negative test case (an input and an oracle). We note that any limitations associated with intrusion detection apply only to the closed-loop system evaluation and not to GenProg in general.

Experimental validity. There exist several threats to the validity of our results. Many of the parameters in the implementation and experimental setup (e.g., Section 5.1) were heuristically chosen based on empirical performance. They may not represent the optimum set of parameter values, representing a threat to construct validity (i.e., we may not actually be measuring a well-tuned genetic algorithm for this domain), although we note that they appear to work well in practice.

Additionally, these parameters, as well as the patterns seen in Figs. 7 and 11, might not generalize to other types of defects or other programs, representing a threat to the external validity of the results. The experiments focus particularly on security-critical vulnerabilities in open-source software, which may not be indicative of all programs or errors found in industry. To mitigate this threat, we attempted to select a variety of benchmarks and

errors on which to evaluate GenProg. More recent publications on the subject of this technique have added several additional benchmarks [35], [46]. We note that such benchmarks are often difficult to find in practice: They require sufficient public information to reproduce an error, access to the relevant source code and revision number, and access to the correct operating environment to enable the reproduction of a given error. Investigating whether the costs reported in Section 6 are similar for other application domains (e.g., **`bind`** or **`openssl`**) and for other types of errors (e.g., time-of-check to time-of-use or unicode parsing problems) remains an area of future research.

8 Related Work

There are several research areas broadly related to the work presented in this paper: automatic bug detection/localization and debugging, automatic error preemption/repair, automatic patch generation, intrusion detection, genetic programming, and search-based software engineering (SBSE).

Research advances in debugging include replay debugging [47] and cooperative bug isolation [1]. Trace localization [48], minimization [49], and explanation [50] 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). However, a narrowed trace or small set of program lines is not a concrete repair. Second, GenProg can theoretically work on any detected fault, not just those found by static analysis tools that produce counterexamples. Finally, these algorithms are limited to the given trace and source code and can thus never localize the "cause" of an error to a missing statement; adding or swapping code to address a missing statement is necessary for many of our repairs. This research can 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. However, a common thread in debugging research is that, while information or flexibility is presented to the developer, repairs for unannotated programs must be made manually.

One class of approaches to automatic error preemption and repair uses source code instrumentation and runtime monitoring to detect and prevent harmful effects from particular types of errors. Demsky et al. [40] automatically insert runtime monitoring code to detect if a data structure ever violates a given formal consistency specification and modify it back to a consistent state, allowing buggy programs to continue to execute. Smirnov et al. [51], [52] automatically compile C programs with integrated code for detecting overflow attacks, creating trace logs containing information about the exploit, and generating a corresponding attack signature and software patch. DYBOC [15] instruments vulnerable memory allocations such that over or underflows trigger exceptions that are addressed by specific handlers.

Other research efforts have focused more directly on patch generation. In previous work, we developed an automatic static algorithm for soundly repairing programs with specifications [31]. Clearview [14] uses runtime monitors to flag erroneous executions, and then identify invariant violations characterizing the failure, generates candidate patches that change program state or control flow accordingly, and deploys and observes those candidates on several program variants to select the best patch for continued deployment. Selected transactional emulation [53] executes potentially vulnerable functions in an emulation environment, preventing them from damaging a system using prespecified repair approaches; a more accurate approach uses rescue points [54]. Sidiroglou and Keromytis [16] proposed a system to counter worms by using an intrusion detector to identify vulnerable code or memory and preemptively enumerated repair templates to automatically generate patches.

These and similar techniques have several drawbacks. First, they require an a priori enumeration of vulnerability types and possible repair approaches, either through the use of formal specifications or the use of external runtime monitors or predefined error and repair templates. In practice, despite recent advances in specification mining (e.g., [55], [56]), formal specifications are rarely available; none of the programs presented in this paper are specified. Moreover, specifications are limited in the types of errors they can find and fix, and cannot repair multithreaded code or violations of liveness properties (e.g., infinite loops). Although some of the nonspecification-based techniques are theoretically applicable to more than one type of security vulnerability, typically, evaluations are limited to buffer over and underflows. The exception to this rule, Clearview, is shown to also address illegal control-flow transfers, but is limited by the availability of external monitors for any given vulnerability type. By contrast, GenProg, designed to be generic, does not require formal specifications or advanced knowledge of vulnerability types and has successfully repaired eight classes of errors to date, including buffer overruns.

Second, these techniques require either source code instrumentation (Smirnov, Demsky), which increases source code size (by 14 percent on Apache in DYBOC), runtime monitoring (DYBOC, Clearview, Keromytis et al., StemSead), or virtual execution (Clearview, selected transactional emulation), imposing substantial runtime overhead (20 percent for DYBOC, up to 150 percent for Smirnov, 73 percent on Apache for StemSead, 47 percent on Firefox for Clearview, and, in general, at least the runtime cost of the chosen monitors). GenProg does not impose preemptive performance or size costs, and minimizes patches as much as possible, though, in theory, generated patches can be of arbitrary size. Our patches are also much more localized than a system that requires system-wide instrumentation and are easily inspected by a human.

Third, these approaches do not evaluate generated repairs for quality or repaired programs for loss of functionality (the Clearview authors note that a manual inspection of their repaired program suggests that functionality is not dramatically impaired). Similarly, they do not evaluate the effect of runtime monitor false positives. While we cannot guarantee correctness, GenProg explicitly encodes testing a patch for correctness with its use of regression tests in fitness evaluation. GenProg produces patches with low overhead in terms of repair time and quality; we have explicitly evaluated the effect of IDS false

positives on system performance and used standard methods to show that they are general.

In 2008, a method for automatically generating exploits from program patches was described [57], generating concern that the method could be used by attackers. Although there are questions about the validity of this threat, it is worth noting that there is no need in our system to distribute a patch. A negative test case can be distributed as a self-certifying alert [58], and individual systems can generate their own repairs.

There is a large literature on intrusion detection for web servers, including anomaly-based methods (e.g., [59]). In principle, many of those techniques, such as those of Kruegel and Vigna [60], Tombini et al. [61], and Wang and Stolfo [62], could be incorporated directly into our proposed closed-loop repair system. Non-webserver programs would require other types of anomaly detection, such as methods that track other layers of the network stack or that monitor system calls or library calls. Other approaches, such as instruction-set randomization [63] or specification mining [64], could also report anomalies for repair. In each of these systems, however, false positives remain a concern. Although Section 6.6 provides evidence that false positives can be managed, a fielded system could incorporate multiple independent signals, initiating a repair only when they agree. Finally, false positives might be reduced by intelligently retraining the anomaly detector after the patch has been applied [65].

Arcuri et al. [20], [66] proposed the idea of using GP to automate the co-evolution of repairs to software errors and unit test cases, demonstrating the idea on a hand-coded example of the bubble sort algorithm. The details of our approach are quite different from Arcuri et al.'s proposal, allowing us to demonstrate practical repairs on a wide variety of legacy programs. Important differences include: We leverage several representation choices to permit the repair of real programs with real bugs, we minimize our high-fitness solution after the evolutionary search has finished instead of controlling "code bloat" along the way, we use execution paths to localize evolutionary search operators, and we do not rely on a formal specifications for fitness evaluation. Several aspects of Arcuri et al.'s work could augment our approach, such as using co-evolutionary techniques to generate or select test cases. However, his work relies on formal specifications, which limits both the programs to which it may apply and its scalability. Orlov and Sipper have experimented with evolving Java bytecode [67], using specially designed operators to modify the code. However, our work is the first to report substantial experimental results on real programs with real bugs. Recently, Debroy and Wong have independently validated that mutations targeted to statements likely to contain faults can affect repairs without human intervention [68].

The field of Search-Based Software Engineering [69] uses evolutionary and related methods for software testing, e.g., to develop test suites [70], [71], [72]. SBSE also uses evolutionary methods to improve software project management and effort estimation [73], to find safety violations [74], and in some cases to refactor or reengineer large software bases [75]. 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.

9 Conclusions

This paper presents GenProg, a technique that uses genetic programming to evolve a version of a program that retains required functionality while avoiding a particular error. We limit the GP search space by restricting attention to statements, focusing genetic operations along a weighted path that takes advantage of test case coverage information, and reusing existing program statements. We use tree-structured differencing techniques and delta-debugging to manage GP-generated dead code and produce a 1-minimal repair. We validate repairs in terms of an input set of test cases.

We used GenProg to repair 16 programs totaling over 1.25 million lines of code and encompassing eight different errors types in 120K lines of program or module code, in 357 seconds, on average; the technique shows encouraging scaling behavior. We evaluated the quality of the generated repairs in the context of a proof-of-concept closed-loop repair system and showed that, for our case-study benchmarks, time lost to the repair process and requests lost to repair quality are both manageable, and in some cases negligible. We showed that IDS false positives similarly represent a manageable threat. Finally, we evaluated our repaired programs on held out test cases, fuzzed inputs, and variants of the original defect, finding that the repairs do not appear to introduce new vulnerabilities, nor do they leave the program susceptible to variants of the original exploit.

We credit much of the success of this technique to design decisions that limit the search space, traditionally a serious difficulty in applying GP to real-world programs. We believe that our success in evolving automatic repairs may say as much about the state of today's software as it says about the efficacy of our method. In modern environments, it is exceedingly difficult to understand an entire software package, test it adequately, or localize the source of an error. In this context, it should not be surprising that human programming often has a large trial and error component, and that many bugs can be repaired by copying code from another location and pasting it in to another, an approach that is not so different from the one described here.

In the short term, GenProg may provide utility as a debugging aid [31] or by temporarily addressing bugs that would otherwise take days to patch or require detrimental temporary solutions, a use-case we explored in our closed-loop repair prototype. In the long term, the technique we have described leaves substantial room for future investigation into the repair of new types of bugs and programs and the effects of automatic repair on program readability, maintainability, and quality. While we remain far from realizing the long-term dream of "automatic programming"—a vision dating back to earliest days of computing—we hope that automatic repair may provide a first step toward the automation of many aspects of the software development process.

Acknowledgments

The authors thank David E. Evans, Mark Harman, John C. Knight, Anh Nguyen-Tuong, and Martin Rinard for insightful discussions. Stephanie Forrest and Westley Weimer gratefully acknowledge the support of the US National Science Foundation (grant CCF-0905236), US Air

Force Office of Scientific Research grant FA8750-11-2-0039 and MURI grant FA9550-07-1-0532, and US Defense Advanced Research Projects Agency (DARPA) grant FA8650-10-C-7089. Stephanie Forrest acknowledges the partial support of CCF-0621900 and CCR-0331580; Westley Weimer acknowledges the partial support of CCF-0954024 and CNS-0716478.

REFERENCES

[1] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, "Bug Isolation via Remote Program Sampling," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 141-154, 2003.

[2] J. Anvik, L. Hiew, and G.C. Murphy, "Coping with an Open Bug Repository," *Proc. OOPSLA Workshop Eclipse Technology eXchange,* pp. 35-39, 2005.

[3] L. Erlikh, "Leveraging Legacy System Dollars for E-Business," *IT Professional,* vol. 2, no. 3, pp. 17-23, 2000.

[4] C.V. Ramamoothy and W.-T. Tsai, "Advances in Software Engineering," *Computer,* vol. 29, no. 10, pp. 47-58, Oct. 1996.

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

[6] M. Jorgensen and M. Shepperd, "A Systematic Review of Software Development Cost Estimation Studies," *IEEE Trans. Software Eng.,* vol. 33, no. 1, pp. 33-53, Jan. 2007.

[7] J. Sutherland, "Business Objects in Corporate Information Systems," *ACM Computing Surveys,* vol. 27, no. 2, pp. 274-276, 1995.

[8] D.E. Denning, "An Intrusion-Detection Model," *IEEE Trans. Software Eng.,* vol. 13, no. 2, pp. 222-232, Feb. 1987.

[9] T. Ball and S.K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Proc. SPIN Workshop Model Checking of Software,* pp. 103-122, May 2001.

[10] D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," *Proc. 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications Companion,* pp. 132-136, 2004.

[11] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-Variant Systems: A Secretless Framework for Security through Diversity," *Proc. USENIX Security Symp.,* 2006.

[12] S. Forrest, A. Somayaji, and D.H. Ackley, "Building Diverse Computer Systems," *Proc. Sixth Workshop Hot Topics in Operating Systems,* 1998.

[13] J. Anvik, L. Hiew, and G.C. Murphy, "Who Should Fix This Bug?" *Proc. Int'l Conf. Software Eng.,* pp. 361-370, 2006.

[14] J.H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M.D. Ernst, and M. Rinard, "Automatically Patching Errors in Deployed Software," *Proc. ACM Symp. Operating Systems Principles,* pp. 87-102, Oct. 2009.

[15] S. Sidiroglou, G. Giovanidis, and A.D. Keromytis, "A Dynamic Mechanism for Recovering from Buffer Overflow Attacks," *Proc. Eighth Information Security Conf.,* pp. 1-15, 2005.

[16] S. Sidiroglou and A.D. Keromytis, "Countering Network Worms through Automatic Patch Generation," *IEEE Security and Privacy,* vol. 3, no. 6, pp. 41-49, Nov./Dec. 2005.

[17] S. Forrest, "Genetic Algorithms: Principles of Natural Selection Applied to Computation," *Science,* vol. 261, pp. 872-878, Aug. 1993.

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

[19] "36 Human-Competitive Results Produced by Genetic Programming," http://www.genetic-programming.com/humancompetitive.html, downloaded Aug. 2008.

[20] A. Arcuri, D.R. White, J. Clark, and X. Yao, "Multi-Objective Improvement of Software Using Co-Evolution and Smart Seeding," *Proc. Int'l Conf. Simulated Evolution and Learning,* pp. 61-70, 2008.

[21] S. Gustafson, A. Ekart, E. Burke, and G. Kendall, "Problem Difficulty and Code Growth in Genetic Programming," *Genetic Programming and Evolvable Machines,* vol. 5, pp. 271-290, Sept. 2004.

[22] D.R. Engler, D.Y. Chen, and A. Chou, "Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. Symp. Operating Systems Principles,* pp. 57-72, 2001.

[23] R. Al-Ekram, A. Adma, and O. Baysal, "DiffX: an Algorithm to Detect Changes in Multi-Version XML Documents," *Proc. Conf. Centre for Advanced Studies on Collaborative Research,* pp. 1-11, 2005.

[24] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" *Proc. Seventh ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 253-267, 1999.

[25] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically Finding Patches Using Genetic Programming," *Proc. Int'l Conf. Software Eng.,* pp. 364-367, 2009.

[26] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues, "A Genetic Programming Approach to Automated Software Repair," *Proc. Genetic and Evolutionary Computing Conf.,* 2009.

[27] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic Program Repair with Evolutionary Computation," *Comm. ACM,* vol. 53, no. 5, pp. 109-116, May 2010.

[28] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer, "Cil: An Infrastructure for C Program Analysis and Transformation," *Proc. Int'l Conf. Compiler Construction,* pp. 213-228, Apr. 2002.

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

[30] B.L. Miller and D.E. Goldberg, "Genetic Algorithms, Selection Schemes, and the Varying Effects of Noise," *Evolutionary Computation,* vol. 4, no. 2, pp. 113-131, 1996.

[31] W. Weimer, "Patches as Better Bug Reports," *Proc. Conf. Generative Programming and Component Eng.,* pp. 181-190, 2006.

[32] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Software Eng.,* vol. 28, no. 2, pp. 183-200, Feb. 2002.

[33] BBC News, "Microsoft Zune Affected by 'Bug'," http://news.bbc.co.uk/2/hi/technology/7806683.stm, Dec. 2008.

[34] B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Comm. ACM,* vol. 33, no. 12, pp. 32-44, 1990.

[35] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing Better Fitness Functions for Automated Program Repair," *Proc. Genetic and Evolutionary Computing Conf.,* 2010.

[36] T. Jones and S. Forrest, "Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms," *Proc. Sixth Int'l Conf. Genetic Algorithms,* pp. 184-192, 1995.

[37] Symantec, "Internet Security Threat Report," http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf, Sept. 2006.

[38] W. Cui, V. Paxson, N. Weaver, and R.H. Katz, "Protocol-Independent Adaptive Replay of Application Dialog," *Proc. Network and Distributed System Security Symp.,* 2006.

[39] K.L. Ingham, A. Somayaji, J. Burge, and S. Forrest, "Learning DFA Representations of http for Protecting Web Applications," *Computer Networks,* vol. 51, no. 5, pp. 1239-1255, 2007.

[40] B. Demsky, M.D. Ernst, P.J. Guo, S. McCamant, J.H. Perkins, and M. Rinard, "Inference and Enforcement of Data Structure Consistency Specifications," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 233-244, 2006.

[41] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms," *Proc. IEEE Symp. Security and Privacy,* pp. 226-241, 2005.

[42] M. Howard and S. Lipner, *The Security Development Lifecycle.* Microsoft Press, 2006.

[43] M. Musuvathi and S. Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs," *Proc. Programming Language Design and Implementation Conf.,* pp. 446-455, 2007.

[44] R.P. Buse and W.R. Weimer, "Automatically Documenting Program Changes," *Proc. Int'l Conf. Automated Software Eng.,* pp. 33-42, 2010.

[45] K. Sen, "Concolic Testing," *Proc. IEEE/ACM 22nd Int'l Conf. Automated Software Eng.,* pp. 571-572, 2007.

[46] E. Schulte, S. Forrest, and W. Weimer, "Automated Program Repair through the Evolution of Assembly Code," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.,* 2010.

[47] L. Albertsson and P.S. Magnusson, "Using Complete System Simulation for Temporal Debugging of General Purpose Operating Systems and Workload," *Proc. Int'l Symp. Modeling, Analysis and Simulation of Computer and Telecomm. Systems,* pp. 191-198, 2000.

[48] T. Ball, M. Naik, and S.K. Rajamani, "From Symptom to Cause: Localizing Errors in Counterexample Traces," *SIGPLAN Notices,* vol. 38, no. 1, pp. 97-105, 2003.
[49] A. Groce and D. Kroening, "Making the Most of BMC Counterexamples," *Electronic Notes in Theoretical Computer Science,* vol. 119, no. 2, pp. 67-81, 2005.
[50] S. Chaki, A. Groce, and O. Strichman, "Explaining Abstract Counterexamples," *Proc. Int'l Symp. Foundations of Software Eng.,* pp. 73-82, 2004.
[51] A. Smirnov and T.-C. Chiueh, "Dira: Automatic Detection, Identification and Repair of Control-Hijacking Attacks," *Proc. Network and Distributed System Security Symp.,* 2005.
[52] A. Smirnov, R. Lin, and T.-C. Chiueh, "Pasan: Automatic Patch and Signature Generation for Buffer Overflow Attacks," *Proc. Eighth Int'l Symp. Systems and Information Security,* 2006.
[53] M.E. Locasto, A. Stavrou, G.F. Cretu, and A.D. Keromytis, "From Stem to Sead: Speculative Execution for Automated Defense," *Proc. USENIX Ann. Technical Conf.,* pp. 1-14, 2007.
[54] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A.D. Keromytis, "Assure: Automatic Software Self-Healing Using Rescue Points," *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 37-48, 2009.
[55] C. Le Goues and W. Weimer, "Specification Mining with Few False Positives," *Proc. 15th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* pp. 292-306, 2009.
[56] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee, "Merlin: Specification Inference for Explicit Information Flow Problems," *Proc. Programming Language Design and Implementation Conf.,* pp. 75-86, 2009.
[57] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation Is Possible: Techniques and Implications," *Proc. IEEE Symp. Security and Privacy,* pp. 143-157, 2008.
[58] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worm Epidemics," *ACM Trans. Computing Systems,* vol. 26, no. 4, pp. 1-68, 2008.
[59] *Recent Advances in Intrusion Detection,* R. Lippmann, E. Kirda, and A. Trachtenberg, eds. Springer 2008.
[60] C. Kruegel and G. Vigna, "Anomaly Detection of Web-Based Attacks," *Proc. 10th ACM Conf. Computer and Comm. Security,* pp. 251-261, 2003.
[61] E. Tombini, H. Debar, L. Mé, and M. Ducassé, "A Serial Combination of Anomaly and Misuse IDSes Applied to http Traffic," *Proc. 20th Ann. Computer Security Applications Conf.,* 2004.
[62] K. Wang and S.J. Stolfo, "Anomalous Payload-Based Network Intrusion Detection," *Proc. Seventh Int'l Symp. Recent Advances in Intrusion Detection,* pp. 203-222, 2004.
[63] W. Hu, J. Hiser, D. Williams, A. Filipi, J.W. Davidson, D. Evans, J.C. Knight, A. Nguyen-Tuong, and J.C. Rowanhill, "Secure and Practical Defense against Code-Injection Attacks Using Software Dynamic Translation," *Proc. Second Int'l Conf. Virtual Execution Environments,* pp. 2-12, 2006.
[64] J. Whaley, M.C. Martin, and M.S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 218-228, 2002.
[65] M.E. Locasto, G.F. Cretu, S. Hershkop, and A. Stavrou, "Post-Patch Retraining for Host-Based Anomaly Detection," Technical Report CUCS-035-07, Columbia Univ., Oct. 2007.
[66] A. Arcuri and X. Yao, "A Novel Co-Evolutionary Approach to Automatic Software Bug Fixing," *Proc. IEEE Congress Evolutionary Computation,* 2008.
[67] M. Orlov and M. Sipper, "Genetic Programming in the Wild: Evolving Unrestricted Bytecode," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1043-1050, 2009.
[68] V. Debroy and W.E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs," *Proc. Int'l Conf. Software Testing, Verification, and Validation,* pp. 65-74, 2010.
[69] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proc. Int'l Conf. Software Eng.,* pp. 342-357, 2007.
[70] K. Walcott, M. Soffa, G. Kapfhammer, and R. Roos, "Time-Aware Test Suitec Prioritization," *Proc. Int'l Symp. Software Testing and Analysis,* 2006.
[71] S. Wappler and J. Wegener, "Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed Genetic Programming," *Proc. Conf. Genetic and Evolutionary Computation,* pp. 1925-1932, 2006.
[72] C.C. Michael, G. McGraw, and M.A. Schatz, "Generating Software Test Data by Evolution," *IEEE Trans. Software Eng.,* vol. 27, no. 12, pp. 1085-1110, Dec. 2001.
[73] A. Barreto, M.D.O. Barros, and C.M. Werner, "Staffing a Software Project: A Constraint Satisfaction and Optimization-Based Approach," *Computers and Operations Research,* vol. 35, no. 10, pp. 3073-3089, 2008.
[74] E. Alba and F. Chicano, "Finding Safety Errors with ACO," *Proc. Conf. Genetic and Evolutionary Computation,* pp. 1066-1073, 2007.
[75] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems," *Proc. Conf. Genetic and Evolutionary Computation,* pp. 1909-1916, 2006.

Claire Le Goues received the BA degree in computer science from Harvard University and the MS degree from the University of Virginia, where she is currently a graduate student. Her main research interests lie in combining static and dynamic analyses to prevent, locate, and repair errors in programs.

ThanhVu Nguyen received the BS and MS degrees in computer science from the Pennsylvania State University and is currently a graduate student at the University of New Mexico. His current research interests include using static and dynamic analyses to verify programs.

Stephanie Forrest received the BA degree from St. John's College and the MS and PhD degrees from the University Michigan. She is currently a professor of computer science at the University of New Mexico and a member of the External Faculty of the Santa Fe Institute. Her research studies complex adaptive systems, including immunology, evolutionary computation, biological modeling, and computer security. She is a senior member of the IEEE and a member of the IEEE Computer Society.

Westley Weimer received the BA degree in computer science and mathematics from Cornell University and the MS and PhD degrees from the University of California, Berkeley. He is currently an associate professor at the University of Virginia. His main research interests include static and dynamic analyses to improve software quality and fix defects.