GenProg: A generic method for automatic software repair

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, Westley Weimer

Abstract—This article 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 1.25M lines of C code and 120K lines of module code, spanning 8 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—D.1.2 Automatic Programming; D.2.7.b Corrections; D.2.5 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% 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 is 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 article describes and evaluates GenProg, (“Genetic Program Repair”), 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 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 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., non-deterministic 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 post-processing 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 article 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%. Across all programs and all trials, we report an average success rate of 77%.
- Experimental 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, non-overflow 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 1.25M lines of code, 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 article 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 motivate important design decisions.

Consider the pseudocode shown in Figure 1(a), adapted from a remote-exploitable heap buffer overflow vulnerability in the nullhttpd v0.5.0 webserver. Function ProcessRequest processes an incoming request based on data copied from the request header. Note that on line 14, the call to calloc allocates 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:

```c
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 ProcessGETRequest). 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.

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

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

Fault localization biases the modifications towards 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 section (b) of Figure 1. A program with this version of ProcessRequest passes both test cases; we call it the primary repair. GP can produce spurious changes to addition to those that repair the program; for example, the search might have randomly inserted return ProcessGetRequest(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.

## 3 Technical Approach

Figure 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 1–5 of Figure 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 pre-determined 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 (AST) that includes all of the statements in the program.

```c
char* ProcessRequest() {
  int length, rc;
  char[10] request_method;
  char* buff;
  while(line = sgets(line, socket)) {
    if(line == "Request:")
      strcpy(request_method, line+12);
    if(line == "Content-Length:"
      length=atoi(line+16);
    }
    if (request_method == "GET")
      buff=DoGETRequest(socket, length);
    else if(request_method == "POST") {
      if (length <= 0)
        return ProcessGetRequest(socket, length);
      buff=DoGETRequest(socket, length);
      rc=recv(socket, buff, length)
      buff[length]="\0";
      return buff;
    }
    return buff;
  }
}
```
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 “(19)”, 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 ⟨ statement, weight ⟩ 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 Figure 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 towards portions of the source code that are likely to affect the bad behavior, while avoiding those that influence good behavior. set_weights(Path_{NegT}, Path_{PosT}) on line 3 of Figure 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.

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 Figure 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 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.** Figure 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 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 statementinserted 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 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

In the case of an insertion or swap, a second statement

In the case of an insertion or swap, a second statement

In the case of an insertion or swap, a second statement

In the case of an insertion or swap, a second statement
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 Figure 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 December 31st, 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 subsections, 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 ReadPOSTData() function, defined in http.c.

```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 is processed and copied:

```c
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 ReadPOSTData. The final, minimized repair is 5 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: non-overflow 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); non-authenticated 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:

```c
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:

```c
516 for (i=1; (char *)p<ber->ber_rwptr; i++) {  
517 tag <<= 8;  
518 if (!(tag & LBER_MORE_TAG_MASK)) break;  
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 /* ?? trip to the resulting binary. */  
524 return LBER_DEFAULT;  
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 multi-byte BER tags. However, only
<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of Code</th>
<th>Code Module</th>
<th>Description</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>22</td>
<td>22</td>
<td>example</td>
<td>infinite loop</td>
</tr>
<tr>
<td>zune</td>
<td>28</td>
<td>28</td>
<td>example [33]</td>
<td>infinite loop†</td>
</tr>
<tr>
<td>uniq utx 4.3</td>
<td>1146</td>
<td>1146</td>
<td>duplicate text processing</td>
<td>segmentation fault</td>
</tr>
<tr>
<td>look utx 4.3</td>
<td>1169</td>
<td>1169</td>
<td>dictionary lookup</td>
<td>segmentation fault</td>
</tr>
<tr>
<td>look srv 4.0 1.1</td>
<td>1363</td>
<td>1363</td>
<td>dictionary lookup</td>
<td>infinite loop</td>
</tr>
<tr>
<td>units srv 4.0 1.1</td>
<td>1504</td>
<td>1504</td>
<td>metric conversion</td>
<td>segmentation fault</td>
</tr>
<tr>
<td>deroff utx 4.3</td>
<td>2236</td>
<td>2236</td>
<td>document processing</td>
<td>segmentation fault</td>
</tr>
<tr>
<td>nullhtpd 0.5.0</td>
<td>5575</td>
<td>5575</td>
<td>webserver</td>
<td>remote heap buffer overflow (code)†</td>
</tr>
<tr>
<td>openldap 2.2.4</td>
<td>292598</td>
<td>6519</td>
<td>directory protocol</td>
<td>non-overflow denial of service†</td>
</tr>
<tr>
<td>cecrypt 1.2</td>
<td>7515</td>
<td>7515</td>
<td>encryption utility</td>
<td>segmentation fault†</td>
</tr>
<tr>
<td>indent 1.9.1</td>
<td>9906</td>
<td>9906</td>
<td>source code processing</td>
<td>infinite loop</td>
</tr>
<tr>
<td>lighttpd 1.4.17</td>
<td>51895</td>
<td>3829</td>
<td>webserver</td>
<td>remote heap buffer overflow (variables)†</td>
</tr>
<tr>
<td>flex 2.5.4a</td>
<td>18775</td>
<td>18775</td>
<td>lexical analyzer generator</td>
<td>segmentation fault</td>
</tr>
<tr>
<td>atris 1.0.6</td>
<td>21533</td>
<td>21533</td>
<td>graphical tetris game</td>
<td>local stack buffer exploit†</td>
</tr>
<tr>
<td>php 4.4.5</td>
<td>764489</td>
<td>5088</td>
<td>scripting language</td>
<td>integer overflow†</td>
</tr>
<tr>
<td>wu-ftp 2.6.0</td>
<td>67029</td>
<td>35109</td>
<td>FTP server</td>
<td>format string vulnerability†</td>
</tr>
<tr>
<td>total</td>
<td>1246803</td>
<td>121337</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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.

about thirty 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 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 Figure 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_create_env function, controlled by the following bounds calculation:

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

The repair modifies this calculation to:

```
off_t weWant = req_cq->bytes_in - offset > FCGI_MAX_LENGTH
weWant = ? 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_replace("A", 65535));
```

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 (65538 "A"’s) with a larger string (65535 "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:
if (Z_STRLEN_P(subject) > 1) {
    Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
    Z_STRVAL_P(search), Z_STRLEN_P(search),
    Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
    php_str_replace_in_subject
}
else if (Z_STRLEN_P(search) > 1) {
    case_sensitivity, replace_count);
    Z_STRVAL_P(result) =
}
else {
    *result = **subject;
    &Z_STRLEN_P(result), case_sensitivity,
    Z_STRVAL_P(replace), Z_STRLEN_P(replace),
    Z_STRVAL_P(result) =
}

w 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 overflow, the underlying problem is a lack of input validation. GenProg operated on the entire wu-ftp 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:

if (Z_STRLEN_P(search) == 1) {
    php_char_to_str_ex(
        Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
        Z_STRVAL_P(search)[0], Z_STRLEN_P(replace),
        Z_STRVAL_P(replace), result,
        case_sensitivity, replace_count);
    } else if (Z_STRLEN_P(search) > 1) {
        Z_STRVAL_P(result) =
    }
}
else {
    *result = **subject;
    rval_copy_ctor(result);
    INIT_PZVAL(result);
    }
     */

5 G 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 Figure 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, non-overflow 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 non-crashing 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 pop_size = 40, which is small compared to typical GP applications; on each trial, we ran the GP for a maximum of ten generations (also a small number). For fitness computation, we set WPosT = 1 and WNegT = 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 $W_{mut}$:

\[
\begin{align*}
W_{path} &= 0.01, W_{mut} = 0.06 \\
W_{path} &= 0.00, W_{mut} = 0.03
\end{align*}
\]

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 ten 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 memoize 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**

Figure 7 summarizes repair results for sixteen 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% 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 sixteen 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 article, 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 multi-byte 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 non-faulty 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 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 pre-sorted. 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), re-initializing the static buffer and avoiding the segfault. Combined with the explanations of repairs for nullhttpd (Section 4.1) and wu-ftp (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-ftp and the availability of its patch.

### 5.3 Scalability and Performance

GenProg is largely CPU-bound. An average repair run took 356.5 seconds. Figure 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% ± 24.0 and negative test cases 32.99% ± 23.17 of the time. In total, fitness evaluations comprise 62.75% ± 30.37 of total repair time. Many test cases include timeouts (e.g., negative test cases that specify an infinite-loop error); others involve explicit internal delays (e.g., ad hoc instructions to wait two seconds for the web server to get “up and running” before requests are sent; the openssl test suite makes extensive use of this type of delay), contributing to their runtime. Compilation of variants averaged 27.13% ± 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 627 seconds for openssl).

Figure 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 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. Figure 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 sub-linearly with search space using other measures. However, the results in Figure 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% 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 vulnerabilities—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 to study repair quality and overhead on programs with realistic workloads.
This section 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 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 (IDS) that detects suspicious HTTP requests based on request features [39]. In a pre-processing 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.

Figure 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.

Several experiments in this section use indicative workloads to measure program throughput pre-, during, and post-repair. We obtained workloads and content layouts from the University of Virginia CS department webserver. To evaluate repairs to the nullhttpd and lighttpd web servers, we used a workload of 138,226 HTTP requests spanning 12,743 distinct client IP addresses over a 14-hour period on November 11, 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 November 11, 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 was 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% utilization (and additional speedups resulted in dropped pack-
<table>
<thead>
<tr>
<th>Case</th>
<th>IDS Class</th>
<th>Repair Made</th>
<th>Behavior In This Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True Neg</td>
<td>—</td>
<td>Common case: legitimate request handled correctly.</td>
</tr>
<tr>
<td>2</td>
<td>False Neg</td>
<td>—</td>
<td>Attack succeeds. Repair not attempted.</td>
</tr>
<tr>
<td>3</td>
<td>True Pos</td>
<td>Yes</td>
<td>Attack stopped, bug fixed. Poor repair may drop requests.</td>
</tr>
<tr>
<td>4</td>
<td>True Pos</td>
<td>No</td>
<td>Attack detected, but bug not repaired.</td>
</tr>
<tr>
<td>5</td>
<td>False Pos</td>
<td>No</td>
<td>Valid request dropped. No repair found.</td>
</tr>
<tr>
<td>6</td>
<td>False Pos</td>
<td>Yes</td>
<td>Valid request dropped. Poor “repair” may drop requests.</td>
</tr>
</tbody>
</table>

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

ets). 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 post- repair 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. Figure 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% 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 Figure 11 quantify 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 Figure 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 multi-character substrings, such as replacing ‘--’ with ‘--’ for strings placed in HTML comments. Our repair leaves multi-character substring behavior unchanged. Many of the other uses of str_replace occur on rare paths. For example, in:

```php
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 subsections 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 Figure 11 shows the results of supplying these requests to each program. Each program failed no additional tests post-repair: for example, lighttpd failed the same 1410 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 inputs.
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 534109 requests from an independent dataset [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 Figure 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 Figure 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. Figure 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 Section 5 and Section 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.

**Non-determinism.** 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 non-interference, that cannot be disproved with a finite number of execution examples; it is not clear how to test or patch non-interference 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% 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% 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 (IDS) 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 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 Figure 7 and Figure 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 of 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 article: automatic bug detection/localization and debugging, automatic error preemption/repair, automatic patch generation, intrusion detection, genetic programming, and search-based software engineering.

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 transactions are evaluated in an emulation environment, preventing them from damaging a system using pre-specified repair approaches; a more accurate approach uses rescue points [54]. Keromytis et al. [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 article 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 non-specification 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 8 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% on Apache in DYBOC), run-time monitoring (DYBOC, Clearview, Keromytis et al., StemSead), or virtual execution (Clearview, selected transactional emulation), imposing substantial run-time overhead (20% for DYBOC, up to 150% for Smirnov, 73% on Apache for StemSead, 47% on Firefox for Clearview and, in general, at least the run-time 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 [20], [66] proposed the idea of using GP to automate the coevolution 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’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’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 Ripper 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 real bugs. Recently, Debroy and Wong have independently validated that mutations targeted to statements likely to contain real bugs. 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.

9 CONCLUSIONS

This article presents GenProg, a technique that uses genetic programming (GP) 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 re-using 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 8 different errors times in 120K lines of program or module code, in 357 seconds, on average; the technique shows encouraging scaling behavior. We validated 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 towards the automation of many aspects of the software development process.

ACKNOWLEDGMENTS

We thank David E. Evans, Mark Harman, John C. Knight, Anh Nguyen-Tuong, and Martin Rinard for insightful discussions. SF and WW gratefully acknowledge the support
REFERENCES


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 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 of 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.

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.