A CLOSED-LOOP REPAIR OF SOFTWARE BUGS

CIS Graduate Student Research Grant Report, 2009

By: **ThanhVu Nguyen**
tnguyen@cs.unm.edu

Department of Computer Science
University of New Mexico
Albuquerque, NM, 87111

Contents

1 Introduction

Detecting and fixing software bugs remains a major burden for the programmers even after the project is released. Despite promising results and efforts in developing automated debugging techniques, these methods still rely on results from rigorous automated testings to locate the errors and require manual modifications from the programmers to fix bugs [3].

We propose an Evolutionary Computing (EC) [5] based approach to automate the task of repairing program bugs in existing software. Programs are evolved and evaluated until one is found that retains the functionality of the original program and fixes the bug that occurred. We first process the source code of the program to produce a path containing traces of execution procedures. This allows us to obtain a *negative* execution path when an error occurs which contains the list of executed statements. Next, our EC algorithm creates new programs by modifying the original code with more bias toward statements that occurred during the negative execution path. Additional tests are incorporated into the fitness function to retain the functionalities of the program.

To accomplish the goal of full automation of the bug repair process, we must automatically detect likely bugs. We propose to use anomaly detection techniques for this component of the project. Anomaly detection uses program execution history to determine what behavior is normal, and thus should be preserved, and what behavior is anomalous, and should be repaired. To be successful, we must detect and repair bugs without foreknowledge and in a timely manner, and repair activities must disrupt services only minimally.

2 Preliminary Work

Our GP program [6, 15] evolves and evaluates program until one is found that retains the functionality of the original program and fixes the bug that occurred. We first process the source code of the program to produce a path containing traces of execution procedures. This allows us to obtain a *negative* execution path when an error occurs which contains the list of executed statements. Next, the GP algorithm creates new programs by modifying the original code with more bias toward statements that occurred during the negative execution path. Additional tests are incorporated into the fitness function to retain the functionalities of the program.

Figure 1 shows a code fragment of the embarrassing bug causing Microsoft Zune media players to freeze on December 31st, 2008 [4]. Throughout this section we refer to this example to to illustrate the important design decisions in our proposed approach. The repair for Zunebug created by our approach is shown in Figure 2.1.

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365)  {
4      if (isLeapYear(year)){
5        if (days > 366)  {
6          days -= 366;
7          year += 1;
8        }
9      }
10     else {
11       days -= 365;
12       year += 1;
13     }
14   }
15   printf("year is %d\n", year);
16 }
```

Figure 1: Zunebug: infinite loop on the last day of leap year (i.e., when the *days* variable has value 366)

2.1 Preprocessing

To obtain the execution path, we use the C Intermediate Toolkit [11] to assign unique ID's to statements in a C program source file. An *execution path* is a record of statement ID's that were called when the program runs against some inputs or *testcases*. We define a *positive* testcase as one that results in expected behavior, e.g., `zunebug(365)`, `zunebug(1000)` and a *negative* testcase as one that causes an error, e.g, `zunebug(366)`, `zunebug(10593)`. One of the key ideas in our approach is to focus on the regions where the error occurs. To do this we assign a weight w to statements occurring in the negative execution path (e.g., lines $1-9$ using the negative testcase `zunebug(366)`). In addition, we also prefer lines that are unique,i.e., only happen in the negative execution path and not on the positive one. To preserve the contents of the original program, we hash its statements into a code repository called C-Bank (Figure 2) and evolve new programs from there.

2.2 Evolution Compution

Our Software Repair using Evolutionary Computing (SREC) algorithm ollows the traditional Evolutionary Algorithm structure. The algorithm maintains a population of chromosomes (programs), selects a pool of individuals based on

Lines	Statement
4	`int isLeapYear(int){ ... }`
6	`days -= 366 ;`
7	`year += 1 ;`
5-8	`if (days > 366){...}`
11	`days -= 365 ;`
12	`year += 1 ;`
4-13	`if isLeapYear(year){...} else{...}`
3-14	`while(days > 365){...}`
2	`int year = 1980;`
15	`printf("current year is %d\n", year);`

Figure 2: Code-Bank for zunebug

their fitness, and modifies them with mutation and recombination operators. The program stops upon reaching a terminating criterion.

The **fitness function** takes a program source code, compiles it, and runs against the set of positive and negative testcases. Finally it returns a score indicating the acceptability of that program. The negative test reproduces the bug in the original program that needs to be fixed and the positive testcases preserve the core functionalities of the program. The fitness score of a program is the weighted sum of the testcases that the program passes. We assign the fitness score of *zero* to programs that do not compile and those with runtime exceeding a preset time threshold (e.g., five seconds).

A subset of the population is **selected** for reproduction using either stochastic universal sampling or tournament selection. Those with fitness scores equal to or less than *zero* are immediately excluded. From here we have a mating pool ready to be modified by the recombination and mutation genetic operations.

Our first **recombination** method adheres to the conventional 1-point *crossover* in EC strategies by exchanging a statement from one parent with another. Our representation of the program allows a statement to contain sub-statements, e.g., conditional and loop code contains all statements within that code block. These statements are chosen uniformly at random regardless of their weights w.

```
1   void zunebug_repair(int days) {
2     int year = 1980;
3     while (days > 365) {
4       if (isLeapYear(year)){
5         if (days > 366) {
6           // days -= 366;//deletes
7           year += 1;
8         }
9         days -= 366;//inserts
10      } else {
11        days -= 365;
12        year += 1;
13      }
14    }
15    printf("year is %d\n", year);
16  }
```

Figure 3: The final Zunebug repair after minimization executes the statement `days -= 366;` when *year* is a leap year, thus guarantees termination to the loop.

Our second implementation called *crossback* preserves the contents of the original program and concentrates on regions in the negative execution path. A single cutoff point c is chosen randomly for both input parents. Then all statements with ID's larger than c are selected for recombination based on their weight values w. The contents of the selected statement s from both parents are replaced by the contents of statement s in the code repository.

We consider each statement in the negative execution path for **mutation** with more bias toward those with heavier weights. The selected statement s is modified with one of the three operations: *delete* the contents of s, *replace* the contents of s with another one from code bank, or *insert* a statement from the code bank after s.

The algorithm **terminates** when an acceptable solution (i.e., one passing all the testcases) is found or it has exceeded the maximum number of preset generations.

2.3 Bloat control and Other Optimizations

Unlike traditional code evolving methods such as Genetic Programming, our EC approach does not suffer from code *bloats* due to several reasons. The algorithm does not add additional nodes (or branches) to existing structure. Inserting a statement j to statement i appends j to i, i.e., $i = \{i;\ j\}$, but does not separate i and j into two distinct nodes. Moreover, SREC evolves programs very similar to the original by limiting the modifications to regions in the negative

Program	Version	Types of Bug	Stmts/ LoC	Pos/Neg Testcases	Negative Path Weight	Success Rate		
						crossover	crossback	size
`zune`	example	Infinite loop	14 / 28	5/2	1.1	58%	71%	4
`uniq`	ultrix 4.3	Segfault	81 /1146	5/1	81.5	100%	100%	4
`look-u`	ultrix 4.3	Segfault	90 /1169	5/1	213.0	100%	99%	11
`look-s`	svr4.0 1.1	Infinite loop	100 /1363	5/1	32.4	100%	100%	3
`units`	svr4.0 1.1	Segfault	240 /1504	5/1	2159.7	5%	7%	4
`deroff`	ultrix 4.3	Segfault	1604 /2236	5/1	251.4	97%	97%	3
`indent`	1.9.1	Infinite loop	2022 /9906	5/1	1435.9	34%	7%	2
`flex`	2.5.4a	Segfault	3635 /18775	5/1	3836.6	4%	5%	3
`atris`	1.0.6	Buffer exploit	6470 /21553	2/1	34.0	82%	82%	3

Figure 4: Experimental Results

execution path and only uses code from the original program. The selection routine also disregards non-working programs. Hence, programs that are not well-formed or deviate greatly from the original have a low chance of being selected. Finally our EC process stops when a candidate passes all the testcases, it doesn't keep evolving to find better solutions.

To improve the performance of our algorithm, we cache the program (its *md5sum* result) and the associated fitness score. Only programs not in the cache are evaluated by the fitness function.

In addition, we apply ideas from structural differencing [1] algorithms and delta debugging [16] to minimize the repair found. Our technique generates a 1-minimal patch that, when applied to the original program, repairs the defect without sacrificing required functionality.

2.4 Experimental Results

Table 4 provides the information on programs that are successfully repaired with SREC. For each program, we run the algorithm for 100 trials using two mutation parameters 0.06 and 0.03. If the first parameter does not work (i.e., gives no repair), the second parameter is used. Column "Stmt/LoC" gives the approximate size of the test programs in terms of number of statements and lines of code. The "Pos/Neg Testcases" column lists the number of positive and negatve testcases used. The "Negative Path Weight" column gives the weighted length of the negative execution path. The last three columns provide statistics when running SREC on these test programs. Columns "crossover" and "crossback" respectively show the success rates of SREC when using different recombination methods. Finally, the "size" column shows the difference in lines of code between the original program and the repair found.

Overall, our algorithm has successfully fixed defects in more than ten programs, including security vulnerabilities in *lighthttpd*, *nullhttpd* (opensource webservers), *openldap* (opensource directory server), and *wu-ftpd* (an ftp server). The success of finding a patch ranges from 4% to 100% with average running time ranging from half a second to ten minutes. The details and statistics of these results can be found in other publications [6, 15, 7].

2.5 Discussions

The two major genetic operators in EC, crossover and mutation, have innovative implementations in our prototype. Our crossover operator crosses individuals in the current generation back with the original (defective) program. Our mutation operator applies several *macro-mutations* to statements along the execution path. In preliminary GP results, we observed that both crossover and mutation operators play an important role in discovering a successful repair. We want to understand in detail what contributes to the success of the GP process. Moreover, our initial designs were based on several hypothesis (e.g., focusing the fixes on statements occurred in execution path is the key to reduce the search space of the algorithm) and we would like to verify and improve upon these thoughts.

2.5.1 The Role of Crossover

Crossover is an important search operator in GP, creating new individuals by recombining partial solutions (subtrees) from individuals. The original implementation (described above) does not take advantage of the potential power of

crossover because individuals are always *crossed back* to the original parent program. In Table 4 we report data comparing the performance of this implementation with a traditional GP crossover operator, which takes two individuals as input, chooses a random position (i.e., statement) from each one, swaps their contents, and returns two new chromosomes. We note that this implementation doesn't take advantage of the statements occured in the negative execution path.

Although the data are not conclusive, the two implementations appear to be comparable: each outperforms the other in some instances. A potential explanation of these results is that crossover is not contributing enough to the search for it to matter, regardless of which version we use.

2.5.2 Varying the Number of Test Cases

The results in Table 4 typically involve six test cases. limiting the fitness function to six discrete values. This could limit the complexity of repair that can be evolved as it provides a relatively coarse signal to GP. Also, programs may have more critical functionality than a few test cases can capture. Typically, programs have too many test cases rather than too few, and test case selection and time-aware test suite prioritization are active research areas (e.g., [14]).

In this section, we ask how GP performance changes when more test cases is used. We did and experiments with 70 distinct trials on the Zune bug, using a fitness function with 24 test cases: 20 positive test cases and 4 negative test cases. represent one standard deviation.

Ideally, the test cases would be independent. In this case they were selected by taking the original five (which were 1000, 2000, 3000, 4000, and 5000) and adding the following: one arbitrary negative number (-100); one negative number that if it were positive would cause the program to hang (-366); one extremely large number (100000000); selecting several arbitrary numbers near leap years and then finding the numbers around those dates that exercise the bugs. The four negative testcases include the original bug (that caused all the zunes to crash in December) as well as several other leap years: 1980 (i.e., day 366), 1984, and 2012.

Unsurprisingly, early generations have fitness values with high variance, and in later generations the variance decreases. The original program passes the positive test cases but fails the negative test cases; it thus has a fitness of 20. Note that over all generations, the average fitness is below the baseline of 20, indicating that the majority of individuals are worse than the original program. Thus, the primary repair is discovered by first losing fitness and then regaining it on the way to the global optimum.

Intuitively, additional test cases could reduce success rate by overly constraining the search space. However, the opposite happened in this example. Using seven test cases, the average success rate is 72%, while the average success rate using 24 test cases is 75%. However, adding test cases does dramatically increase the total running time of the algorithm: with seven test cases, the average time to discover the primary repair is 56.1 seconds; with 24, this time increases to 641.0 seconds. This makes sense: Every fitness evaluation potentially involves running all of the test cases. Therefore, in general, we prefer a fitness function with a small number of test cases.

3 Closed-Loop System

To accomplish the goal of full automation of the bug repair process, we must automatically detect likely bugs. We use anomaly detection techniques for this part of the research. Anomaly detection uses program execution history to determine what behavior is normal, and thus should be preserved, and what behavior is anomalous, and should be repaired. To be successful, we must detect and repair bugs without foreknowledge and in a timely manner, and repair activities must disrupt normal services only minimally.

We restrict the scope of this part with security vulnerabilities of *webservers*, an area where defects are common and can have serious consequences. For the experiments, we adopted an intrusion-detection system (IDS) developed by Ingham et al[9, 10], which uses a probabilistic finite state machine approach to detect suspicious HTTP requests. The alphabet of the state machine includes a set of tokens, or features, describing a request (e.g., hostname, file path, client IP address). In the training phase, legitimate requests are used to construct a finite state machine model of normal requests. Ideally, the language of the machine includes all legitimate requests and no malicious ones.

Once the model has been constructed, it is then used in the testing phase to classify new HTTP requests. Intuitively, a new request is labeled as normal if it is in the language of the state machine and is labeled suspicious if it is rejected by the state machine. State machine path compression and generalization are used to improve performance and the models tolerance of new requests. In addition, edge weights allow the model to produce probabilistic outputs rather than a binary classification.

Detected anomalies are treated as negative testcases and passed to our repair system. We focus our experiments on the open-source webservers including Lighttpd. In this section we give details on our methods and experimental results.

3.1 Experiments Setup

Training data: We first collect common webserver bugs (e.g., code injection, non-control data attacks, DOS), starting with the collections provided by[10] which contains over 65 attacks spanning 8 operating system and 13 different servers. In addition, we obtained workloads from the University of Virginia Computer Science department webserver. To evaluate repairs to the basic webservers, (nullhttpd and lighttpd) we used an indicative 14-hour period from November 11, 2008 involving 138,226 HTTP requests spanning 12,743 distinct client IP addresses.

We trained the IDS system on 534109 requests collected from various websites. This training data set has not been filtered and might contain anomalous requests, and thus the model may mistakenly accept the similar anomalous requests. The training process, which need only be done once, took 528 seconds on a machine with quad-core 2.8 GHz and 8GB of RAM. The resulting system assigned a score to each incoming request ranging from 0.0 (definitely anomalous) to 1.0 (definitely normal). On our testing workload, the IDS has no false negatives with a threshold of 0.02: it successfully detects the publicly available *nullhttpd* and *lighttpd* exploits below as anomalous and does not flag any other requests. Details of the exploits and repairs for nullhttpd and lightted are listed in next section.

3.2 Detecting and Repairing Security Exploits

3.2.1 lighttpd exploit: heap buffer overflow

```
POST / HTTP/1.0
Content-Length: -800

\195\171
--netric--1\195\1281\195\1551\195\1371\195\146... // more shell code
```

This exploit in Lighttpd mod_fastcgi[1] attempts to retrieve the contents of /etc/passwd to the attacker. The problem lies in the *fcgi_env_add* routine, which uses *memcpy* to assign data to a buffer pointer without proper bounds checks.

A proper fix would be do a check on memory violation before calling the memcpy function, however recall that our repair system does only reuses code in the existing program, thus if there is no application memory checking code in the program, then this fix won't be used. Instead the repair modifies a bound check inside *fcgi_create_env* (which calls the *fcgi_env_add* routine), changing the statement:

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

This effectively prevents the large data from being sent to the script interpreter process causing the bufer overflow while still allowing the good (positive) requests to go through. Generating the repair took less than a minute with the repair system.

3.2.2 nullhttpd exploit: remote heap buffer overflow

```
conn[sid].PostData=calloc(conn[sid].dat->in_ContentLength+1024, sizeof(char));
pPostData=conn[sid].PostData;
...
do {
  rc=recv(conn[sid].socket, pPostData, 1024, 0); /* buffer overflow! */
  ...
  pPostData+=rc;
} while ((rc==1024)||(x<conn[sid].dat->in_ContentLength));
```

[1] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-4727

This public exploit in Nullhttpd v.0.5.0 [2] attempts to give the running server a buffer overflow which then crashes the system. The problem lies in the *ReadPostData()* routine where the value in *in_ContentLength* is supplied by the attacker which causes *conn[sid].PostData* to overflow. Fortunately there is another place in the code where this Post-data copy has proper memory bound check thus allows the repair system to reuse (with some additional modifications). Generating the repair takes approximately four minutes which more than half of the time are used to exercise the testcases to make sure the modified code still retain its original functionalities.

3.3 Realtime Deployment

In the architecture proposed in Section 4, the repair process generates an entirely new executable. In practice, it might be desirable to apply a patch to a running executable without halting and restarting the system. The dynamic patching problem is well-studied [2], and many existing solutions are applicable here. In a security setting, however, software dynamic translation (SDT) systems such as STRATA [13] can be used to perform the patching. In such a system, all instructions are translated and stored in a run-time code cache before being executed; the SDT can be directed to flush relevant portions of the cache (on a per-function basis) and refill them from the code segment of a second binary image. Software dynamic translation dovetails particularly well with our approach, since it can be used to detect certain intrusions (e.g., via low-overhead instruction set randomization [8]), record the statements visited for the weighted path, sandbox the variants for testing [12] and deploy the resulting repair. Although we have not yet tested run-time repairs, we are working on it and are optimistic that the setup and implementation will be available soon.

4 Summary

After more than 30 years of research in software engineering and programming languages, software is still to a large extent developed, debugged, maintained, and tested by humans. At the same time, the size of our software base continues to grow, the complexity of the environments in which software runs has increased dramatically, and we expect ever greater functionality out of our software. As a consequence, software today is in many ways less reliable and more prone to bugs than it was a decade ago.

While there are many approaches for automatically detecting and locating security vulnerabilities, less atten- tion has been paid to their automatic repair. Our work give a preliminary implementation of a closed-loop architecture for repairing real security vulnerabilities in off-the-shelf security-critical applications. Our basic repair technique is based on genetic programming. It uses a weighted path to restrict modifications to areas of the program likely to contain the bug. It uses positive and negative test cases to encode required functionality and to demonstrate the attack. We combine this automated repair with anomaly detection to turn attacks in to negative test cases, allowing programs with standard regression tests to be repaired automatically. We empirically evaluated, on large indicative workloads applied to full systems, the cost of crafting the repair, the costs of repairs that sacrifice functionality, and the costs of anomaly detection false positives that lead to unneeded repairs. Our results indicate that low-quality repairs caused by insufficient test suites are a cause for concern, while false positive repairs typically have negligible effect. Our primary result, however, is the demonstration of six automatic repairs on 100K lines of server code for five types of security vulnerabilities: remote heap buffer overflow, non-overflow denial of service, format string vulnerability, local stack buffer overflow, and integer overflow.

[2] `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2002-1496`

References

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

[2] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. Opus: online patches and updates for security. In *USENIX Security Symposium*, pages 19–19, 2005.

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

[4] BBC News. Microsoft zune affected by 'bug'. In *http://news.bbc.co.uk/2/hi/technology/7806683.stm*, Dec. 2008.

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

[6] S. Forrest, W. Weimer, T. Nguyen, and C. L. Goues. A genetic programming approach to automated software repair. In *The Genetic and Evolutionary Computation Conference*, 2009.

[7] C. L. Goues, T. Nguyen, W. Weimer, and S. Forrest. Using execution paths to evolve software patches," search-based software testing. In *Search-Based Software Testing*, 2009.

[8] 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. In *Virtual Execution Environments*, pages 2–12, 2006.

[9] K. Ingham. *Anomaly detection for HTTP intrusion detection: Algorithm comparisons and the effect of generalization on accuracy*. PhD thesis, Univ. of New Mexico, Albuquerque, NM, 2007.

[10] K. L. Ingham, A. Somayaji, J. Burge, and S. Forrest. Learning dfa representations of http for protecting web applications. *Computer Networks*, 51(5):1239–1255, 2007.

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

[12] K. Scott and J. W. Davidson. Safe virtual execution using software dynamic translation. In *Computer Security Applications Conference*, pages 209–218, 2002.

[13] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Symposium on Code generation and optimization*, pages 36–47, 2003.

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

[15] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, 2009.

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