CAREER: Scalable and Trustworthy Automatic Program Repair

A fundamental challenge for computer scientists over the next decade is to produce and maintain systems that have fewer defects and are more resilience to attacks. Software maintenance accounts for over $70 billion per year and is focused on repairing defects. Defects are reported so rapidly that programs ship with known bugs and even security bugs take 28 days, on average, to fix.

Intellectual merit. The proposed research will reduce the time and effort gaps between finding and fixing software defects by producing a scalable and trustworthy technique to automatically repair program bugs. Preliminary results demonstrate that genetic programming, program analysis and test cases can generate viable repairs for off-the-shelf C programs. Once a defect has been found, evolutionary algorithms produce variant programs, performing a biased search to locate one that fixes the defect without impairing key functionality. Both the defect and the program's requirements are represented using standard test cases. Prototype tools using this approach have repaired eight classes of defects in 18 programs, including multiple security-critical servers, totaling over 170,000 lines of code. There are multiple components to the proposed research, which ultimately aims to apply automated repair to real systems by addressing the challenges of scalability and trust:

- **Design and investigate algorithms for scaling automated repair to real-world programs:** The research will focus on programs with large test suites and difficult-to-localize defects. Using the insight that only the final repair need be perfect, intermediate steps are permitted to "break the rules" (e.g., by using only a portion of the test suite or by guessing a fault location) as long as the final repair is fully vetted. The research will investigate test suite selection, change impact analysis, and fault localization within the context of repair.
- **Study and evaluate approaches for producing trusted repairs:** The research will produce high-quality repairs that can be directly verified. The primary insight is that trustworthy repairs must use processes that developers already trust, such as existing specifications or invariants. The research will incorporate partial correctness specifications, specification mining, typestate-guided program repair, and specification-guided program synthesis.
- **Propose and carry out compelling empirical repair evaluations:** The research will produce methods for gaining confidence that a produced repair does not introduce new defects, does not impair normal functionality, and generalizes to address the root of the problem. This will include evaluating repairs on indicative workloads as well as against random fuzz inputs and mutated attacks. In addition, the research includes an automated hardening scenario in which the program and attacks against it are coevolved over time, simulating parts of the security arms race.

Broader impact. Billions of dollars are lost each year due to software defects, and the proposed research is well-suited to addressing multiple classes of real-world bugs. The educational goals of this project are to foster student interest in the exciting area of automated program repair. First, we will provide hands-on research experience and mentoring for both graduate and undergraduate students. The results of the research will be integrated into existing graduate courses and will help to create challenging new undergraduate courses in the area of programming languages. Third, we will continue our fellowship and outreach activities to encourage participation from underrepresented undergraduates.

Key Words: software; repair; formal specification; fault localization; test suite; fuzz testing.

1 Introduction

Software quality is an overarching problem. Software maintenance, traditionally defined as any modification made on a system after its delivery, sums up to $70 billion per year in the US [97], and can account for as much as 90% of the total cost of a typical software project [87]. Repairing defects and otherwise evolving software are major parts of those costs [79]. We propose to reduce the costs associated with debugging by repairing programs automatically.

Currently, manually repairing bugs is a major bottleneck to software quality. First, bugs are *plentiful.* The number of outstanding software defects typically exceeds the resources available to address them [6]. For example, in 2005, one Mozilla developer claimed that, "everyday, almost 300 bugs appear [. . .] far too much for only the Mozilla programmers to handle" [7, p. 363]. Second, bug repair is *expensive.* In a 2008 survey, 139 North American firms each spent an annual mean of $22 million fixing software defects [14]. Moreover, the cost of repairing a bug increases as development progresses: a $25 fix while the program is under development can increase to $16,000 after the software has been deployed [112]. Third, bug repair is *time-consuming.* On the Mozilla project between 2002 and 2006, half of all fixed bugs took developers over 29 days [44]. For the ArgoUML and PostgreSQL project, half of all bugs required over 190 days to fix. This is particularly troubling in critical code: in 2006, it took 28 days on average for maintainers to develop fixes for security defects [98]. Fourth, as a result, *deployed* programs contain costly known and unknown bugs [58]. In a 2008 FBI survey of over 500 large firms, the average annual cost of computer security defects alone was $289,000 [82, p.16]. In 2002, NIST calculated the average US annual economic cost of software errors to be $59.5 billion, or 0.6% of the GDP [73].

To alleviate this burden, we propose an automatic technique for repairing program defects. Our technique targets off-the-shelf legacy applications. We use genetic programming to evolve program variants until one is found that both retains required functionality and also repairs the defect in question. To our knowledge, this is the first successful use of genetic programming to repair real applications, and our approach has been well-received critically (see Section 2.4). Our technique takes as input a program with a detected defect, a set of positive testcases that encode required program behavior, and a failing negative testcase that demonstrates the defect. We may also take advantage of partial correctness specifications, test case generators, and indicative workloads, if present, to help generate and validate repairs. In contrast to previous repair approaches, which handle only a specific type of defect such as buffer overruns [89, 90], our technique is generic, and we have applied it successfully to multiple types of defects from the realms of software engineering and security.

Our preliminary experiments hold out the promise of automated repair for real systems, and the proposed research program aims to fulfill that promise. The two key challenges are scalability and trust. Consider a critical server that is subject to a security attack at midnight. If the attack can be detected, our proposed approach can fashion a repair that can be deployed until developers arrive in the morning to manually address the situation. To be useful in such a situation, automated program repair must produce trustworthy patches for large systems in a timely manner. Two insights guide this research. First, we note that only the final repair has to be perfect, and we thus permit intermediate steps to temporarily "bend some rules" (e.g., use only some of the test cases, or make a guess about the location of the fault) as long as ultimate result is fully correct. This insight provides the freedom to scale to large systems. Second, we claim that the key to trustworthy repairs is to use processes that developers already trust, such as existing specifications or fuzz testing. This insight encourages us to take advantage of existing verification information, such as by using partial specifications to rapidly rule out certain repairs.

In addition to evaluating our results in terms of the resources required to produce trusted repairs to known defects, we also propose an "automated hardening" evaluation scenario. In this setting we simulate both sides of the security "arms race" by co-evolving a program and attacks against it. If a program snapshot from six months ago can be automatically hardened to defeat an attack from three months ago with no knowledge of that attack, we gain confidence that our technique can be used proactively as well as reactively.

2 Automated Program Repair — Preliminary Results

This section describes the technical basis of our proposal: the use of genetic programming and test cases to evolve repairs for program defects. Preliminary results published this year give additional details [37, 70, 109].

2.1 Technique

Genetic programming (GP) is a computational method inspired by biological evolution which discovers computer programs tailored to a particular task [55]. GP maintains a population of individual programs. Computational analogs of biological mutation and crossover produce program variants. Each variant's suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. GP has solved an impressive range of problems (e.g., see [1]), but to our knowledge it has not previously been used to evolve off-the-shelf legacy software.

We use genetic programming to discover program variants that avoid a given buggy behavior, such as a security vulnerability. A program variant is represented by its abstract syntax tree (AST). The fitness, or aptitude, of a variant is based on whether or not it passes the test cases. Formally, a *test case* consists of input to the program, any necessary global state, and an *oracle comparator* function that encodes the desired response (e.g., known-good output from a previous version) [18]. We use the term *positive test case* to refer to a standard test case (e.g., regression, unit, etc.) that encodes desired program behavior. The term *negative test case* refers to a program input that demonstrates the defect. The negative test case must include an oracle comparator that can decide if the program exhibits the defect: it is thus similar to a *self-certifying alert* [26].

We guide program modifications by restricting attention to program statements that are likely to have caused the defect. We define the *weighted path* for a negative testcase as the sequence of statements visited during the execution of the negative testcase but not that of a given positive testcase. We hypothesize that statements in the weighted path are more likely to cause the defect than statements that are executed for both the positive and negative testcases (e.g., initialization code). We assume that the latter statements are less likely to contribute to the defect and that statements only visited during the unsuccessful run are more likely to relate to the defect in question.

We apply two sorts of program modifications to statements along the weighted path. First, our *mutation* operator deletes, inserts or replaces statements. When inserting a new statement, we choose a statement that is already present in the program and copy it to a new location, hypothesizing that the program contains the seeds for its own repair. That is, we assume that the program is usually implemented correctly, and that the defect is a behavioral outlier [33]. For example, if the defect is caused by a missing null check, the program is likely to contain a correct null check somewhere else that can be adapted to form the repair. Second, our *crossover* operator combines two program variants, potentially swapping statements along their weighted paths.

The fitness of a variant is computed by pretty pretting and compiling its modified AST and running the test cases against the resulting executable in a safe sandbox. The *fitness function* is simply the weighted sum of the total number of positive and negative test cases passed. Genetic programming is an iterative process. As such, variants with high fitness values are retained into the next *generation*, while low-fitness variants are discarded. A program variant that passes all the test cases is called a *primary repair*, and the iterative search stops when a primary repair is found. Using tree-structured differencing [2], we can view the primary repair as a set of changes to be applied to the original program. We use *delta debugging* [117, 119] to efficiently compute a one-minimal subset of changes from the primary repair. A change subset is one-minimal if removing even a single change from it would cause the resulting program to fail a test case. This minimization technique eliminates redundant code from the primary repair to produce the *final repair*. Because the GP technique operates on the AST and source code of the program, the repair can be reified as a standard `patch` (i.e., as produced by `diff`) or deployed directly.

```
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        else {
10       }
11     }
12     else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf("year is %d\n", year);
18 }
```

(a)

```
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; // deleted
7          year += 1;
8        }
9        else {
10       }
11       days -= 366; // inserted
12     } else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf("year is %d\n", year);
18 }
```

(b)

Figure 1: Buggy Zune code (a) and produced repair (b).

2.2 Motivating Example

On December 31st, 2008 a widely reported bug was discovered in the Microsoft Zune media players, causing them to freeze up [17]. The fault was caused by a bug in the code [64] shown in Figure 1(a). When the value of the input `days` is the last day of a leap year (such as 10593, which corresponds to Dec 31, 2008), the program enters an infinite loop on lines 3–16. While the highlighted code may seem relatively simple, over two million players were affected by the defect [24].

We now walk through the evolution of a repair for this program. We first produce its AST and determine the weighted path, using line numbers to indicate statement IDs. The positive test case `zunebug(1000)` visits lines 1–8, 11–18. The negative test case `zunebug(10593)` visits lines 1–16, and then repeats lines 3, 4, 8, and 11 infinitely. For this example, our negative test cases consist of the inputs 366 and 10593, which cause an infinite loop (instead of the correct values, 1980 and 2008), and our positive test cases are the inputs 1000, 2000, 3000, 4000, and 5000, which produce the correct outputs 1982, 1985, 1988, 1990 and 1993.

For the purposes of illustration, we focus on one variant, V, that starts out identical to the original program. In Generation 1, two operations mutate V: the conditional statement "`if (days > 366) { days -= 366; year +=1; }`" is inserted between lines 6 and 7 of the original program; and the statement "`days -= 366`" is inserted between lines 10 and 11. Note that the first insertion includes not just the `if` but its entire subtree. This produces code that passes the negative test case 366 (year 1980) and one positive test case 1000.

V survives Generations $2, 3, 4$ and 5 unchanged, but in Generation 6, it is mutated with the following operations: lines 6–10 are deleted, and "`days -= 366`" is inserted between lines 13 and 14.

At this point, V passes all of the test cases, and the search terminates, producing V as the initial repair. The minimization step is invoked to discard unnecessary changes. Compared to the original program (and using the line numbers from the original), there are three key changes: c_1 = "`days -= 366`" deleted from line 6; c_2 = "`days -= 366`" inserted between lines 9 and 10; and c_3 = "`days -= 366`" inserted between lines 10 and 11. Only c_1 and c_3 are necessary to pass all tests, so change c_2 is deleted. This produces the final repair, shown in Figure 1(b). This is one of the many possible repairs that the search might produce.

2.3 Preliminary Results

We have implemented a prototype tool that uses the proposed repair algorithm to fix defects in off-the-shelf, unannotated, legacy C programs [37, 70, 109]. Figure 2 lists preliminary results of eighteen repairs on over 170,000 lines of code. Minimizing the primary repair using delta-debugging is a deterministic post-processing step that takes twelve seconds (eight fitness evaluations) on average. The final result is a standard `diff-`

Program	LOC	Program Description	Defect Repaired	Time	# Fitness
`gcd`	22	Euclid's algorithm	infinite loop	276 s	909
`zune`	28	MS Zune example	infinite loop	78 s	460
`uniq`	1146	text processing	segfault	32 s	139
`look-ultrix`	1169	dictionary lookup	segfault	42 s	120
`look-svr4`	1363	dictionary lookup	infinite loop	51 s	42
`units`	1504	metric conversion	segfault	1528 s	9014
`deroff`	2236	document processing	segfault	132 s	227
`nullhttpd`	5575	webserver	heap buff. overflow [74]	1394 s	1800
`openldap io`	6159	authentication server	non-overflow D.O.S. [76]	665 s	8
`leukocyte`	6718	computational biology	segfault	544 s	12
`indent`	9906	source code processing	infinite loop	7614 s	13628
`python complexobject`	11227	web app. interpreter	overflow error	1393 s	23222
`lighttpd fastcgi`	13984	webserver CGI module	heap buff. overflow [60]	395 s	52
`imagemagick fx`	16851	image processing	incorrect image output	1240 s	131
`flex`	18775	scanner generator	segfault	4660 s	9560
`atris`	21553	graphical game	stack buff. overflow [10]	84 s	285
`php string`	26044	web app. interpreter	integer overflow [78]	2678 s	18081
`wu-ftpd`	35109	FTP server	format string vuln. [113]	5397 s	74
total	179369	(18 distinct programs)	(18 defects in 8 classes)	1567 s	4320

Figure 2: Preliminary Experimental results: eighteen programs automatically repaired by our prototype tool. The "Defect" column describes the bug; a citation indicates that publicly-reported security vulnerability was repaired using the published exploit as the negative test. "Time" gives the sum total seconds to generate the primary repair, including time spent in the genetic algorithm (typically 10% of the listed time), time spent compiling variants (30%), and time spent calculating the fitness of a variant by running the test suite (60%). Fitness calculations dominate the run-time; the final column gives the total number of times the entire test suite was run to generate and validate a successful repair. In each case, at most ten positive test cases were used. Each row represents the average of 100 random trials.

style patch, two to eleven lines long on average for our benchmarks, that can be presented to developers or incorporated directly.

The programs repaired span a wide variety of domains and include operating system utilities, graphical games, webservers, and scientific computing applications. The entire program can be repaired as a unit (e.g., as in `wu-ftpd`), or a single module can be repaired in isolation without modifying or analyzing the rest of the code (e.g., as `php`, where only the `string` module was repaired). The defects repaired fall into two broad categories. First, we have repaired many standard software engineering bugs, including infinite loops (e.g., `indent`), segfaults and crashes (e.g., `leukocyte`), and logic errors resulting in incorrect output (e.g., `imagemagick`'s `fx` module). We have also used generated repairs for five common categories of security vulnerabilities: remote heap buffer overflow, non-overflow denial of service, format string vulnerability, local stack buffer overflow, and integer overflow. Those vulnerabilities were repaired in standard security-critical programs such as `openldap`, `wu-ftpd`, `lighttpd` and `php`. We view security as a particularly important domain for this technique, since helping developers rapidly patch security flaws would be beneficial. We thus particularly value our approach's ability to repair multiple different types of security vulnerabilities (i.e., not just buffer overruns) without prior knowledge or a model of the attack or desired repair. To the best of our knowledge, this is the first generic, source-level repair technique to work across multiple attack types and many programs (cf. [61, 83, 89, 90, 91, 92, 93]).

However, significant challenges remain. One impediment for a genetic programming evolutionary algorithm is the vast search space it must sample to find a correct program. In Figure 2, the time taken to repair a program is not related to its size in lines of code, and while many repairs are quite rapid (e.g., under a minute), some may be infeasible for many settings (e.g., over 13000 test suite evaluations for `indent`). In addition, although all of the repairs pass all of the test cases and have been manually inspected, developers may be wary of using them. While merely presenting a developer with a candidate patch reduces the time required to address the bug [103], additional assurance is required before such repairs would be automatically deployed. The heart of this proposal is research to scale this technique to more demanding systems while producing trustworthy repairs.

2.4 Academic Reception

The preliminary work described here and published this year has been well-received by the academic community, winning two best paper awards (ICSE'09 [109] and GECCO'09 [37]), a best short paper award and best presentation award (SBST'09 [70]), the 2009 IFIP TC2 Manfred Paul Award for "excellence in software: theory and practice" [77, €1024], and the gold medal at the 2009 "Humie" awards for human-competitive results produced by genetic and evolutionary computation [54, $5000]. We are quite excited about this line of research and perceive that some others are as well. This critical validation and attention provides some confidence that the tools and algorithms we produce may be adopted and disseminated if they can scale to produce trustworthy repairs.

3 Proposed Research

While our preliminary results demonstrate that our technique can automatically repair multiple classes of defects for programs or modules of moderate size, conversations with developers and suggest that two major hurdles remain before our repairs would be used in industrial practice, even in an advisory capacity: *trust* and *scalability*. Regarding trust, even if a repair produced by a biased search passes all available test cases, additional guarantees about how the process works and how the repair behaves are necessary if the repair is to be deployed. As just one example, developers may seek assurance that a repair generated to fix one security flaw does not introduce a different security flaw. Regarding scalability, while our prototype implementation can produce viable repairs a small number of test cases, in practice mission-critical systems have quite large test suites. Since program size is not the bottleneck, we must demonstrate that we can produce repairs for such systems.

Ultimately, trust and scalability are intertwined in this setting: if test cases are the primary method for ensuring that our repairs do not impair functionality, scaling to larger test suites will be necessary to ensure trust in repairs. Similarly, many trust-enhancing techniques reduce the search space of possible repairs (e.g., by removing from consideration repairs that violate partial specifications) and thus may potentially increase scalability. Two main insights guide our proposed research. First, we can allow an intermediate repair to be only partially validated as long as the final repair is fully vetted. Second, we must use techniques that developers already trust, such as fuzz testing and partial correctness specifications, to guide repair generation and verify repairs.

We propose two research directions related to increasing trust in the repairs:

1. Guide repair construction using **formal specifications**. We will combine synthesis, typestate repair, and refactoring for verification to produce verifiable patches.

2. Evaluate repair quality empirically with **fuzz tests**, **workloads** and **hardening**. We will gain confidence that functionality is not impaired, that new defects are not introduced, and that repairs are general rather than fragile.

And two research directions related to producing repairs for large programs:

3. Produce repairs in the presence of **large test suites**. We might, for example, use a subset of the test suite for intermediate fitness evaluations, scaling without sacrificing repair quality.

4. Produce repairs of **poorly-localized** faults. When the weighted path is does not localize the defect's location, we will reduce the search space by interleaving fault localization with program repair.

3.1 Research: Quality: Guiding Repairs with Formal Specifications

Formal, machine-checkable specifications of a software program hold great promise for creating trustworthy repairs. Unfortunately, formal specifications are difficult to apply directly to program repair. For example, a program verified against a full functional specification either has no bugs, has a bug in its specification, or used an unsound verification technique — the latter two cases are beyond the scope of this research. However, programs often have *partial* correctness specifications. For example, many programs developed at Microsoft, including all of Windows Vista, contain relatively simple specifications and annotations [27] related to the correct use of locks, in-bounds array accesses, and other typestate properties [11, 28, 43]. In this context, a formal specification is often a first-order logic Floyd-Hoare pre- or post-condition or assertion. However, it may also take the form of a finite state machine over an alphabet of important operations [103]. Partial correctness specifications permit the possibility that a "partially correct" program still exhibits a bug. We can ensure that our fixes for such bugs do not introduce a specification violation. In this light, a testcase is just a partial correctness specification that says "this input should lead to this observable behavior."

We propose to construct repairs that conform to such partial specifications. A first step would be to use the existing verification process as a final test case. For example, if a candidate repair fixed the bug under consideration, but introduced a violation to the locking specification, it would be rejected, and the genetic programming search would continue. At a deeper level, we further propose to use the specification to rule out infeasible mutations, and thus reduce the search space, while maintaining repair trustworthiness. For partial specifications that can be phrased at the program-expression level (e.g., pre- and post-conditions [115, 116], or the type state or locking policies mentioned above), potential modifications would be subject to symbolic execution [11, 22, 28] or verification condition generation [57].

In cases where logical pre- and post-conditions are available, we propose to restrict attention to repairs for which the resulting formulas are at least as strong as they are in the original program. Consider a program that has an array bounds specification and maintains the invariant $0 \leq i \leq 10$ on a particular path. A candidate repair that maintains $2 \leq i \leq 7$ would be acceptable, since $2 \leq i \leq 7 \Rightarrow 0 \leq i \leq 10$, but a repair that maintains $0 \leq i \leq 12$ would not be, since $0 \leq i \leq 2 \not\Rightarrow 0 \leq i \leq 10$ (e.g., $i > 10$ may correspond to an out-of-bounds array access). A modification can be rejected in this manner with a single query to an automated theorem prover [29, 31] rather than an expensive set of testcase evaluations.

If pre- and post-conditions are available for a region of interest, we may synthesize conformant code rather than using a genetic programming search. Recent work on verification and synthesis by others has been able to generate non-trivial algorithms (e.g., implementations of merge sort) given standard pre- and post-conditions [95, 96, 99] (e.g., the output array is a permutation of the input array, and time is bounded by $\Theta(n \log n)$). In previous work, we developed an automatic algorithm for soundly repairing programs when formal typestate specifications are present [104]. The algorithm adjoins multiple copies of the finite state specification and uses those copies to track missing or extraneous events. By tying the edges to program locations, a provably safe repair can be generated along a single path. By using a sound path predicate analysis [84] or including dynamic path tracking in the repair [12], a single-path repair can be correctly applied to a larger program. Finally, we also have previous work in transforming programs to make them easier to verify with respect to pre- and post-conditions [115, 116]. For example, although a loop and its unrolling have equi-satisfiable verification conditions, those conditions may not be equally easy for existing model checkers and theorem provers to verify. We have found that by applying a series of semantics-preserving transformations to *reduce* efficiency, we can make a program easier to verify. If our automated repair is correct but cannot be verified directly, this transformation approach can be used to refactor that repair into one that can be verified.

We propose to combine the approaches of synthesis, correct typestate repair, and refactoring for verification, to produce verifiable patches. If our weighted path (or another fault localization technique) can narrow down the location of the fault to a portion of the code for which unsatisfied pre- and post-conditions

Program	Successful Requests Before Repair	Repair Made?	Requests Lost to Repair Quality	Fuzz Test Failures Generic	Fuzz Test Failures Exploit
`nullhttpd`	100.00%	Yes	$0.00\% \pm 0.25\%$	$0 \mapsto 0$	$10 \mapsto 0$
`lighttpd`	100.00%	Yes	$0.03\% \pm 1.53\%$	$1410 \mapsto 1410$	$9 \mapsto 0$
`php`	100.00%	Yes	$0.02\% \pm 0.02\%$	$3 \mapsto 3$	$5 \mapsto 0$
False Positive 1	100.00%	Yes	$0.00\% \pm 2.22\%$	$0 \mapsto 0$	—
False Positive 2	100.00%	Yes	$0.57\% \pm 3.91\%$	$0 \mapsto 0$	—
False Positive 3	100.00%	No	—		—

Figure 3: Empirical security repair evaluation. Each row represents a different repair scenario, and is separately normalized to 100%. The number after ± indicates one standard deviation. "Lost to Repair Quality" indicates the fraction of the daily workload lost after the repair was deployed. "Generic Fuzz Test Failures" counts the number of held-out fuzz inputs failed before and after the repair. "Exploit Failures" measures the number of held-out fuzz exploit variants failed before and after the repair.

already exist (e.g., as part of a partial correctness specification), we can generate candidate repairs by deleting subsets of the affected lines and using the synthesis procedure fill in the blanks with code that satisfies the specification. If a typestate specification already exists, our previous work can be used to generate a correct repair. In either case, the resulting repaired program can be verified using the same mechanism as the original (e.g., model checking, theorem proving, etc.), and our refactoring approach can be used to phrase the repair so as to facilitate and make practical that verification.

If pre- and post-conditions are not available, we can use invariant generation techniques or specification mining to infer them [5, 72, 108, 114]. For example, if the original program has the postcondition X for the code region of interest on all positive test cases, and the postcondition $X \wedge \neg Y$ on the negative test case, we can synthesize code to meet the postcondition $X \wedge Y$ as a candidate repair. In practice, finding similar clauses in first-order logic formulas may require a large number of queries to an automated theorem prover, although many optimizations from the domain of predicate abstraction may be applicable [11]. In addition, if specification mining is used to guide the synthesis, accurately inferring specifications without false positives will be critical. We have recent work in automated specification mining that improves on the state of the art by reducing the false positive rate by an order of magnitude [38].

3.2 Research: Quality: Empirical Evaluations of Repair Quality

In current practice, many potential repairs, especially in security-critical domains, are subjected to a barrage of adversarial tests to gain confidence in their correctness. We propose to subject our repairs to a similar battery of tests to empirically evaluate their correctness. We seek confidence that the repair has not impaired key functionality, has not introduced new vulnerabilities, has generalized to repair the defect in some depth, and that repairs based on mistaken defect identification do not harm the system. To that end, we propose four methods to evaluate the correctness of repairs generated by our previous technique: held-out indicative workloads, generic fuzz testing, targeted fuzz testing, and "false positive" repairs.

The first evaluation uses a large indicative workload that is not used during the repair precess. Unless the nature of the repair makes it impossible, the repaired program should return all of the same answers on this workload, bit for bit, as the original program, and should do so in the same amount of time or less. The second evaluation uses fuzz testing [65] or some similar technique to generate held-out adversarial inputs or otherwise attack the program. The repaired program should not be vulnerable to any new attacks or demonstrate any additional failure modes as compared to the original program. The third evaluation uses fuzz testing or manual annotation to modify the negative test case or exploit to create similar attacks using the same vector. For example, a remote buffer overrun exploit might be changed by inserting dummy instructions into the payload; our repair is successful if it also defeats such mutant attacks. The final evaluation tests the repair technique in the environment by controlling for the repair itself: additional "repairs" are generated with normal behavior on innocuous input given as the negative test case. This simulates what would happen in the case of an intrusion detection false positive or other misclassification of the actual defect.

We have performed preliminary experiments along these lines for repairs to two webservers and a web application interpreter. We used a held-out workload of 138,226 HTTP requests spanning 12,743 distinct client IP addresses. Figure 3 shows the results for the four evaluations. For each of the three repairs, the deployed, repaired program effectively responded to all requests correctly (the standard deviation is higher than the absolute number of lost requests because the systems benchmarking process is not deterministic, and the repaired version performs better than the original on some runs; that is, any change is in the noise). Microsoft requires that security-critical changes be subject to 100,000 fuzz inputs [47] (i.e., randomly-generated structured input strings). 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. We applied them to the webservers before and after the repair: the "Generic" column in Figure 3 shows the results. For example, `lighttpd` failed the exact same 1410 fuzz tests before and after the repair, suggesting that the repair may not have introduced new failure modes. Similarly, 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 our repairs introduce any errors that were detected by 100,000 fuzz tests, and in every case our repairs defeated variant attacks based on the same exploit, showing that the repairs were not simply fragile memorizations of the input.

Finally, we randomly selected three normal requests from the held-out workload to serve as false positives and instructed the system to "repair" `nullhttpd` against them. False positive #3, a standard `HTTP GET` request, was not repairable using our algorithm: there was no way to rule it out while satisfying a positive test case for `GET index.html`. The other two false positives did lead to "repairs": for example, #2 was a `HEAD` request related to caching information, and the "repair" disabling caching, causing additional requests from clients. However, the positive tests cases again prevent any significant impairment on the indicative workload, and neither of the "repairs" introduced any detectable security vulnerabilities.

We hypothesize that repairs generated by our technique will not impair normal behavior, introduce new errors, or be fragile memorizations of the negative input. Our preliminary results provide some support for this on three programs, but much remains to be done. This research thrust will evaluate that hypothesis directly, as well as allowing for controlled experiments with repair quality as the dependent variable. For example, we will attempt to determine how many and what sort of positive test cases are necessary for a high quality repair, and how much post-repair evaluation is necessary to achieve a given level of confidence that a repair is not faulty.

3.3 Research: Scalability: Test Suite Selection for Program Repair

In practice, many important programs have a vast suite of test cases. Regression testing dominates portions of maintenance and development [79], accounting for as much as half of the cost of software maintenance [52, 86]. While our repair technique has been quite successful at producing high-quality repairs using less than a dozen positive test cases to encode program requirements, we wish to scale to repair programs with hundreds or thousands of test cases. Since our default fitness function evaluates a variant program on the entire test suite, we propose to change how our algorithm measures the fitness of a variant.

Program	#TC	Time (sec)	# TC Runs
`zune`	100	12 $\mapsto$ 10	460 $\mapsto$ 106
`uniq`	100	62 $\mapsto$ 8	715 $\mapsto$ 89
`look-ultrix`	100	65 $\mapsto$ 9	545 $\mapsto$ 94
`deroff`	100	71 $\mapsto$ 15	915 $\mapsto$ 73
`lighttpd`	500	90 $\mapsto$ 73	3580 $\mapsto$ 362
`nullhttpd`	200	168 $\mapsto$ 87	3200 $\mapsto$ 201
`imagemagick`	145	2216 $\mapsto$ 543	1715 $\mapsto$ 178
`leukocyte`	300	5527 $\mapsto$ 205	13205 $\mapsto$ 362

Figure 4: Preliminary test suite reduction experimental results. The "#TC" column gives the number of positive testcases used for the repair. The two remaining columns list the change in performance with our technique applied.

Running test suites is an expensive proposition in standard software engineering, and many techniques have been developed to reduce the cost of testing while maintaining its benefits [42, 63].

We propose to investigate and adapt two techniques to the context of program repair: *change impact analysis* [8, 56, 81] and *test suite prioritization* [85, 86, 101].

In change impact analysis, program analysis techniques are used to determine which tests must be rerun after a modification to a program. For example, if the effects of a change can be definitively localized to module A, unit tests for module B need not be rerun. We propose to use change impact analysis on the population of program variants we maintain: if a variant was evaluated on a set of testcases in a previous generation, and only a few lines in it were changed in the interim, it is probably not necessary to rerun each of those tests. In our preliminary results, the average number of statement level changes between fitness evaluations was around two, suggesting that change impact analysis could be easily applied. Similarly, in some cases, the fitness of a variant only changes in as few as a third of all generations [37, Fig.1], suggesting that up to two-thirds of tests may be run unnecessarily. We thus propose to extend our internal representation to explicitly track previously-established test case results on a per-variant basis. We can then use a conservative change impact analysis to invalidate the results and rerun the tests as necessary. This can reduce repair times without changing the resulting repair. In essence, change impact analysis will allow us to soundly cache test case evaluation results, decreasing the number of tests that need to be rerun.

In test suite prioritization, a subset of the available test suite is run in a specified order with the typical goal of detecting defects or gaining confidence as quickly as possible. Existing techniques range from greedy approaches (e.g., choose test cases in order of decreasing coverage) to time-aware approaches based on knapsack solvers [101]. The structure of our repair problem provides an advantage not normally available: the subset of tests used on any particular variant need not be perfectly indicative, as long as any candidate repair is subject to the entire test suite. We thus propose a basic hybrid algorithm in which we avoid expensive coverage calculations and instead choose a random subset of the test suite each time a variant is evaluated. A variant that passes every test in that random subset is then evaluated against the entire test suite. We have implemented this technique, and preliminary results suggest that it is a slight improvement over existing approaches for our domain, but, more importantly, that it can dramatically reduce the total time taken to find a repair. On eight different programs and over 1500 test cases, our preliminary approach reduces the total number of test case evaluations by 90% on average; Figure 4 shows these results. The figure also includes the cost of running any candidate repair that passes its subset of tests on the entire test suite. Programs that were previously the slowest to repair show large gains: `leukocyte`, with 300 test cases, takes three minutes, instead of ninety, to discover a viable repair.

3.4 Research: Scalability: Repairing Poorly-Localized Faults

Running the test suite to evaluate program variants dominates the running time of our repair algorithm. Since we apply mutations only along the weighted path (our approximation to where the fault may be located), the length of that path controls the size of state space of nearby programs explored to find a possible repair. A large weighted path that contains many irrelevant statements will result in the generation of many fruitless variants, each one of which must be run against the test suite before being discarded.

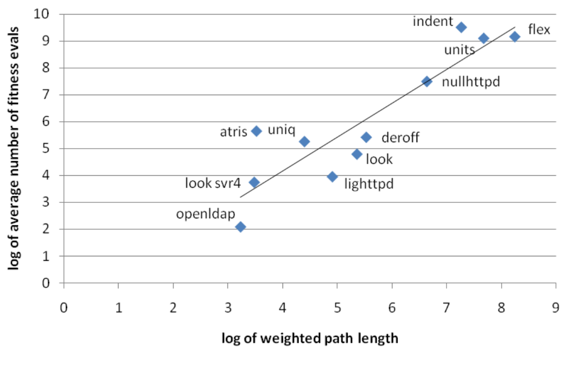

Figure 5: Program repair time scales with weighted execution path size (averaged over 100 runs, note log-log scale).

Figure 5, plots weighted path length against search time, measured as the average number of fitness evaluations before the initial repair is discovered. On a log-log scale, the relationship is roughly linear with slope 1.26 (90% confidence: [0.90, 1.63]). Although this preliminary work is not conclusive, the plot suggests that search time may scale as a power law of the form $y = ax^b$ where b is the slope of the best fit line (1.26) and $b = 1$ would indicate that search time grew linearly. This suggests that search time grows as a small polynomial of the weighted execution path and not as an exponential.

We thus propose to scale our technique to repair more programs by taking advantage of existing techniques from the domains of fault localization and path slicing. In this context, *fault localization* takes a program and a defect and returns a subset of locations in the program related to the defect [39, 40, 13]. For example, if a program has a locking bug and no locking behavior ever depends in any way on the value of a variable `x`, then statements that only assign to `x` can be removed from consideration. Recent advances may dramatically reduce the size of the weighted path, and thus the search space. For example, Jhala and Majumdar report that the largest of 313 paths from the `gcc` Spec95 benchmark can be sliced from 82,695 nodes to 43 nodes [50]. Unfortunately, many localization techniques cannot be applied directly without additional input: Jhala and Majumdar require a single program point that encapsulates the error, while Ball *et al.* require a function mapping program states to success or failure judgments. In essence, many such techniques require foreknowledge of what the fault *is* before they can narrow down what is relevant to it. If the program we are repairing fails the negative test case because of a null pointer dereference, division by zero, assertion failure, or other precise exception, we can use that program point or expression directly.

In general, however, the program will fail the negative test case by producing the wrong output, rather than by crashing in an obvious manner. While we do not know the cause of the bug, we pay no penalty beyond time for an incorrect guess. We can thus use fault localization as a form of hypothesis testing: rather than attempting to holistically repair the program as a unit, we can test the individual hypotheses that various statements are the root cause. If the size of the weighted path is P, in the worst case we can guess each of the P locations in turn as an important program point, apply fault localization, and attempt a repair. If the wrong program point was chosen, the repair attempt will fail. Back-of-the-envelope calculations suggest that this approach will reduce overall repair times in common situations. For example, if the running time of our algorithm is P^b and fast fault localization reduces path length by a factor of L, then this approach will decrease the overall running time when $P \times (P/L)^b < P^b$, i.e., when $P < L^b$. For $L = 1000$ [50] and $b = 1.26$ (Figure 5), the approach could be useful for $P < 6000$, a path size almost twice as long as the longest we have encountered to date.

4 Proposed Experiments

Each of the hypotheses in the proposed research above has an associated set of experiments. The list below serves as a basic experimental plan; we will adapt the plan as the research progresses. The preliminary work published this year [37, 70, 109] will serve as a **baseline algorithm** to which new repair techniques can be compared. The programs described in Figure 2 serve as a set of **default benchmarks**, although we will continually seek out new defects to repair. The number of seconds and test case evaluations required to produce a repair, as well as the evaluations described in Section 4.1, will serve as **default metrics** by which repairs or repair techniques can be evaluated. For example, a repair produced by interleaved fault localization can be compared directly to any repair produced by the baseline algorithm using those metrics. Beyond shared use of the Empirical Evaluation research thrust, all of the proposed experiments are independent, contain their own local metrics, and can be pursued in any order.

4.1 Experiments: Quality: Empirical Evaluations of Repair Quality

This research thrust proposed four key evaluations: held-out indicative workloads, generic fuzz testing, targeted fuzz testing, and “false positive” repairs. In addition to using those metrics as described in our preliminary work (see Figure 3), we further propose an additional *automatic hardening* scenario related to defects and program revisions. We will iteratively apply the repair technique to past versions of programs, creating synthetic diversity [71, 111] and preemptively defending against unknown defects.

Consider a program that encountered a defect (e.g., a security attack) six months ago. We propose to take the program and its test suite from nine months ago and iteratively repair it with no knowledge of the held-out six-month defect. If a sequence of repairs to a far-past version of a program causes it to become immune to a near-past bug, we gain confidence that iterative repairs to a present program may cause it to become immune to future bugs.

Our automated repair approach requires a negative testcase or other notion of the defect to be repaired. For this evaluation scenario we can use any software quality metric as a signal. We propose to evaluate FindBugs defect reports [46] and fuzz testing [65]: given a program with a test suite and no known bugs, we will treat any error report from a bug-finding static analysis or any crash when the program is applied to randomly-generated input as a defect to be repaired. In addition, we propose to co-evolve the attack and the repair. Four of the six security attacks in Figure 2 were presented as C programs that, when compiled and run, attacked a running server. We propose an iterative process in which the program is evolved to resist the attack, and then the attack is evolved until it can again defeat the program (simulating some parts of the standard security "arms race").

While this technique is easily to parallelize and CPU time is relatively cheap, the computational burden may prove prohibitive. In essence, the technique fails if it would take two weeks of automatic hardening to make a program immune to an attack that will arrive in one week. However, it is also more general than standard diversity techniques such as address space randomization [88] or instruction set randomization [15, 16, 48] because it can produce new content or control flow, and thus potentiallydefeat a larger class of attacks.

We hypothesize that in the majority of cases where our technique can repair a defect at all, it can be used to automatically harden a program against that defect in less time than it took for the defect to arise naturally. We will select defects at random from projects with linked bug and source code repositories, or otherwise consider defects for which we know repairs are possible (see Figure 2). For each defect, we will iterate over multiple possible time windows (i.e., starting X weeks before the defect with a budget of X weeks for hardening) and run multiple random trials for each window, measuring the percentage of hardened variants that ended up immune to the held-out bug.

Whenever possible, repairs and repair techniques from other research thrusts will be evaluated using time and test cases required to produce a repair, held-out indicative workloads, generic fuzz testing, targeted fuzz testing, and "false positive" repairs, and the automatic hardening scenario.

4.2 Experiments: Quality: Guiding Repairs with Formal Specifications

We can evaluate the proposed research against the baseline algorithm, which does not consider specifications. When partial correctness specifications are present, one additional metric is the chance of generating a repair that fixes the bug while respecting all available specifications. A second metric is the time taken to find a repair that fixes the bug while respecting other specifications. We hypothesize that incorporating knowledge of specifications into the repair process will allow for the construction of specification-conformant repairs in at most as much time as it takes the vanilla approach to generate non-conformant repairs (e.g., since querying an automated theorem prover takes less time than running a test suite to to rule out a candidate repair).

Additional evaluation will be required if specification mining is used to guide program synthesis to fashion correct-by-construction repairs. First, the combined technique is unlikely to apply to all repairs: if either the specification mining or the synthesis fails, no repair will be generated. We will measure the fraction of instances for which this occurs on all available repairable programs, and characterize the situations under which the technique can succeed. Even if a candidate repair is produced that passes all available tests, a lack of precision in the specification miner may result in a repair with unintended side-effects. We will evaluate the repairs using the metrics described in Section 4.1. We hypothesize that an approach that combines specification mining (or manually-added partial correctness specifications) with program synthesis can produce repairs without unwanted side-effects for certain classes of programs.

4.3 Experiments: Scalability: Test Suite Selection for Program Repair

We hypothesize that we can dramatically reduce the number of test case evaluations required to automatically repair multiple classes of defects, even in the presence of large test suites. Our primary metrics are the average number of test case evaluations and seconds required to obtain a successful repair. Change impact analysis should apply to any of our existing repairs without changing the behavior or repair quality; instead, it will reduce the number of test cases run at the expense of additional computational overhead. We hypothesize that in that vast majority of cases, the overall repair time will be reduced, and we will categorize those cases for which the technique is ineffective. For test suite prioritization, we will continue our preliminary experiments, and additionally measure the efficacy of hybrid test selection on manually-created and automatically-generated test cases.

4.4 Experiments: Scalability: Repairing Poorly-Localized Faults

We hypothesize that for a large class of repairs (in particular, for the largest weighted paths we have encountered), fault localization can be used in conjunction with automatic program repair to reduce the total repair time. We will evaluate this hypothesis directly, first on programs with crashing failures (e.g., failed assertions) to which fault localization can be directly applied, and second when fault localization and program repair are used to test fault hypotheses. The key metric will be the average number of fitness evaluations and total time required to produce a repair of equivalent quality (although since both will use the same test suite, a large quality difference is not expected).

5 Background

Related work falls into four broad categories: research to improve manual debugging, efforts to recover from defects at run-time, automatic repairs for particular defects, and genetic programming.

In general, debugging involves locating and repairing a defect by hand. Recent advances include replay debugging [3], the analysis of postmortem crash dumps [62], and cooperative bug isolation [59] In all of these approaches, although additional information or flexibility is presented to the developer [118], the full repair must always be generated manually.

Demsky *et al.* [30] present a technique for data structure repair. Given a formal specification of data structure consistency, they modify a program so that if the data structures ever become inconsistent, they can be modified back to a consistent state at runtime. Their technique does not modify the program source code in a user-visible way. Instead, it inserts run-time monitoring code that “patches up” inconsistent state so that the buggy program can continue to execute. Finally, their technique only targets errors that corrupt data structures — it is not designed to address the full range of logic errors. Many of the programs we have repaired or propose to repair are outside the scope of their technique.

Rinard *et al.* [83] present a notion of *failure-oblivious computing*, in which compiler-inserted bounds checks discard invalid memory accesses, allowing servers to avoid security attacks. Boundless memory blocks store out of bounds writes for subsequent retrieval for corresponding out of bounds reads. Techniques also exist for eliminating infinite loops. The overhead of these approaches varies, memory errors are the primary consideration, and a localized patch is not produced. Similarly, Smirnov *et al.* [92, 93] automatically compile C programs with integrated code for detecting overflow attacks, creating trace logs that contain information about the exploit, and generating a corresponding attack signature and software patch. This technique is restricted to buffer overflows and introduces as much as 150% run-time overhead by adding additional code into the original source program. In contrast to both of these approaches, we propose to produce explicit, low-overhead patches for multiple types of defects.

Keromytis *et al.* [90] propose a system to counter worms through automatic patch generation. Their technique requires an *a priori* enumeration of possible attack and repair types, and is heavily dependent on an intrusion detector to identify vulnerable buffers. While their work is theoretically applicable to more than one type of security vulnerability, they focus entirely on the detection and repair of buffer over- and under- flows and envision handling additional vulnerabilities by crafting new repair modules for specific vulnerability types. A key distinction between our proposal and the work of Keromytis *et al.* is that our approach does not require advanced knowledge or profiles of vulnerability types to craft a repair.

6 Broader Impact and Education Plan

The overall goal of this work is to reduce the cost of defect repair and thus improve software development. The proposed research must thus be accompanied by efforts on additional fronts over the next five years. First, we must mentor and educate upcoming researchers and engineers. This includes the design of new courses as well as efforts to involve members of varied backgrounds in the research. Second, we must make it easy and compelling for current developers to make use of the technology we develop and the practices we propose. We will demonstrate the utility of our work through case studies and freely-available tools instantiating our techniques.

6.1 Education and Mentoring

Graduate Curriculum. The University of Virginia includes some of the world's leaders in the fields of Software Engineering, Sensor Networks, Virtual Execution Environments, Graphics, Modeling and Simulation, and Security. We will work closely with these researchers in the development of tools and techniques for software repair that are relevant to their domains. We will encourage this collaboration directly by extending the graduate Programming Languages course to cover recent cross-domain advances. In addition, we believe that the attractive force of faculty collaboration is mediated by particles known as graduate students: we will require graduate class projects that integrate programming language analyses and techniques with software from other domains. For example, the `leukocyte` program used in preliminary work is the result of such a collaboration with local architecture researchers who are using it to investigate manycore coprocessors [20]. Similarly, we have begun to work with local graphics researchers on "repairing" vertex and pixel shader programs to improve their efficiency, possibly by altering non-human-discernible visual output. This sort of collaboration gives the students firsthand experience with the challenges of analyzing real-world code, and those challenges are often the greatest spurs to new techniques.

Undergraduate Curriculum. We will redesign the undergraduate Programming Languages class so that it includes recent techniques for improving software quality via language-based approaches. Relevant techniques include using CCured [25], Cyclone [51] and Tal [66] to produce low-level systems software safely, as well as more recent techniques like Mace [53] for building distributed systems and StreamIt [100] for building streaming programs. In addition, the course will give examples of the transition from research to practice, such as that of Pizza into Java Generics [75]. These changes are near and dear to our heart, as the current undergraduate PL class is somewhat of a cleverly-disguised compilers course, and since undergraduate compilers is also offered, plenty of scope exists for injecting language-based techniques. We will also design a new undergraduate course:

> ***Course Development:*** **Program Analysis Tools (Undergraduate).** This class will cover techniques and tools for analyzing software, giving students a grand tour of many modern techniques. Software model checkers [11, 22, 43], bug-finding tools [23, 28, 32, 41, 46, 107], and memory management aids such as Valgrind [69] will play key roles. In addition, invariant detectors [36, 72] and specification miners [4, 5, 33, 108, 110] will be emphasized, especially as they relate to trusted program repair (Section 3.1). Students will begin by experimenting with malware and attacks, from fuzz [65] to code injection, to motivate the need for reliable and secure programs and for the empirical evaluations in Section 4.1. The course lectures will cover the techniques the tools are based upon as well as comparisons of their domains and relative strengths and weaknesses. The course project will ask students to use one of the tools presented in class to find a vulnerability in a large open-source program, devise an attack against that defect, and then design a defense against the attack. As this project progresses, we will incorporate an assignment to evaluate generated repairs for such defects.

Diversity. We are continually interested in mentoring undergraduates by actively involving them in the research program, and the three undergraduates in this research group reflect that. We have been particularly involved in encouraging women to consider and pursue computer science at the graduate and undergraduate level. We have worked with the local chapter of the ACM Committee on Women in Computing (ACM-W) to help it transition from involving mostly graduate students to incorporating undergraduates as well. Meetings that we sponsored and advertised drew in eleven additional undergraduate women on average, doubling local participation.

In addition, we have used unrestricted industrial funding (augmented by the award money described in Section 2.4) to create an annual merit scholarship for undergraduate women interested in computing and communication [105]. The scholarship specifically targeted undergraduate women who might not otherwise have considered computer science. In its three years the scholarship has received dozens of application materials covering five U.S. states (IN, MD, PA, TX, and VA) and multiple non-CS academic disciplines (e.g., Environmental Science, International Studies, Multimedia Instructional Design). Each applicant was required to consider and describe how computer science could relate to her field and future. The scholarship prize includes stipend money to support undergraduate research, and winners are encouraged to work on undergraduate research. The scholarship was sufficiently popular that the department chair was willing to provide matching funds, increasing the number of undergraduate women we are able to recognize for excellence and encourage to consider and pursue computer science.

Mentoring. We will involve undergraduates heavily in pursuing this research agenda through undergraduate thesis projects. Much of the preliminary work for this project was carried out by undergraduates. For example, Ethan J. Fast and Briana Mae Satchell produced all of the data in Figure 4. Previous undergraduate researchers in the group have published and gone on to top-tier CS grad schools (e.g., Nicholas Jalbert [49] to Berkeley in '08, Elizabeth Soechting [94] to Wisconsin-Madison in '09). We are also interested in working with undergraduates officially advised by other faculty members, and that work has led to publication (e.g., Adrienne Felt [34]).

6.2 Case Studies and Tools to Promote Adoption

In a large sense, the overarching goal of this proposal is to encourage the adoption of our automated repair technique by making it scale to rapidly produce trustworthy repairs. We thus believe that developers will be more likely to adopt this technique after seeing the generated repairs vetted multiple ways, or after seeing the "automatic hardening" scenario preemptively repair an unknown bug or a repair automatically validated against an existing specification. We have already made our tools available to researchers on demand, and plan a general open source release in the near term. We have a track record of successfully making tools available in projects like CCured [68] and Cil [67]. There is thus no secondary plan to promote adoption: it is an integrated goal.

7 Results from Prior NSF Support

Westley Weimer. *CT-ER: Automatic Identification and Protection of Security Critical Data* (PI: D. Evans, CNS-0627623). This completed project aimed to develop a general and practical defense against future non-control data attacks by automatically transforming programs to protect critical data. It led to multiple publications [19, 34, 45, 106] as well as national media attention directly mentioning our female undergraduate researcher [9, 102]. *CT-T: Practical Formal Verification By Specification Extraction*, (PI: J. Knight, CNS-0716478). This ongoing project aims to develop a new approach to formal verification in which a mid-level specification is extracted from low-level program annotations. The program is shown to implement the mid-level specification and the mid-level specification is shown to imply the high-level specification; the resulting two-step verification process is more practical than a monolithic approach [21, 115, 116].

In addition, the PI has a pending NSF grant *SHF: Medium: Collaborative Research: Fixing Real Bugs in Real Programs Using Evolutionary Algorithms* (CCF-0905373; Co-PI: S. Forrest, submitted 10/31/2008). That pending grant proposes to investigate issues such as new genetic algorithm operators, incorporating anomaly detection to point out defects, conducting human studies to evaluate repairs, collecting templates of likely repairs, and repairing multithreaded programs — none of which are proposed here. The two proposals focus on different aspects of the broad realm of automated program repair, and none of the research described here is mentioned in the other proposal except for the study of test suites (which merits only two paragraphs there, and is explicitly listed as not in the core of that research proposal).

8 Summary and Long-Term Vision

If succesful, the proposed research will reduce the time and effort gaps between finding and fixing software defects by making automatic repair feasible and trustworthy for large systems. Currently, although tools such as delta debugging and fault localization exist, program repair remains a largely manual process. We propose to lift much of that burden.

The research and implementation work proposed here will form the core of the dissertations of two students. One will focus on repair quality, the other on scaling automated repair to real systems. We can thus make progress on each front independently, and allow the students to work together while keeping separate, clearly delineated goals.

Year 1. Incorporate collaboration into grad course. Implement empirical evaluation metrics (Section 4.1). Release public tools.

Year 2. Begin research and design repair with formal specifications (Section 3.1) and test suite selection for repair (Section 3.3).

Year 3. Evaluate formal specification (Section 4.2) and test suite selection (Section 4.3) thrusts. Begin work on automatic hardening (Section 3.2) and interleaving fault localization with repair (Section 3.4).

Year 4. Develop program analysis tools undergrad course. Evaluate automatic hardening and fault localization thrusts (Section 4.4).

Year 5. Integrate program repair into undergrad class. Finish evaluating formal specification, test selection and localization research thrusts using all empirical metrics including automatic hardening (Section 4); conclude case studies.

The proposed research touches on a number of challenging areas and will be a strong source of research problems for years to come. Software quality is a key problem that affects most computing disciplines. We have more defects than we know what to do with [7], many of them go unfixed [58], and both repair and the consequences of failing to repair carry a large cost [14, 112, 82, 73]. Software maintenance is dominated by such bugs [79], which traditionally require developer time and effort to fix [80, 44, 98]. We envision a future in which automatic bug repair is a trusted, scalable reality for many classes of defects. The proposed research will take tangible, concrete steps toward the ultimate goal of automating software maintenance.