Automatic, Efficient, and General Repair of Software Defects using Lightweight Program Analyses

Ph.D. Dissertation Proposal

Claire Le Goues
legoues@cs.virginia.edu

September 27, 2010

1 Introduction

Software errors are both pervasive and expensive. Mature software projects ship with both known and unknown bugs [49], and the number of outstanding defects typically exceeds the resources available to address them [5]. A 2008 FBI survey of over 500 large firms reported that security defects alone cost such firms $289,000 annually [66, p.16]; a 2002 NIST survey calculated that software errors in the US cost 59.5 billion (0.6% of GDP) annually [59]. Manual error repair remains the norm, but has become unsustainably time-consuming and expensive. A 2008 survey of 139 North American firms found that each spent an average of $22 million annually fixing software defects [10]. Many defects, including critical security defects [78], remain unaddressed for long periods of time [36] despite the fact that the cost of repairing a defect can increase dramatically as development progresses [85].

As a result, *automatic error repair*, in which an algorithm constructs a repair to an error without human intervention, has become urgent. However, previous approaches suffer from several key deficiencies. Most trade generality for correctness by considering only a predefined set of vulnerability and repair types; most repair memory overflow errors exclusively [71, 74]. Lack of generality reduces the utility of such techniques in the current landscape, where new types of vulnerabilities regularly gain prominence [35]. Additionally, existing repair techniques often prohibitively increase either code size [18] or run-time [70] or both [63], precluding applicability to the real-world, legacy systems where they are needed the most.

We propose to automatically repair errors by using search algorithms, in conjunction with existing test cases and lightweight program analyses, to find a version of a program that does not exhibit a given error but that still maintains required functionality and correctness. An algorithm based on search admits a more general approach to automatic repair because it is not definitionally constrained to one type of runtime monitor or repair template. Unfortunately, previous efforts to apply search algorithms to automatic error repair have been stymied by scalability concerns [32], which must be overcome for such a technique to be successful.

Our proposed approach relies on two key insights to achieve generality and scalability. First, **test cases provide useful insight into program behavior**. While they provide weaker guarantees than formal correctness proofs, test suites as measures of correctness scale more readily to real-world systems [48, 76]. They are also readily available in practice and easier to generate than formal behavioral specifications [28, 68]. Moreover, test cases can flexibly encode a broad set of behaviors, and program behavior on test inputs can provide considerable feedback about dynamic program semantics. We propose that an automatic repair technique that relies on test cases can apply to a general set of both programs and error types. Second, **existing program behavior contains the seeds of many repairs**. Fundamentally, we presume that most programmers program correctly most of the time: incorrect program behavior, such as a forgotten null check, is often correctly implemented elsewhere in the source code of the same program. This insight underlies certain static specification mining techniques [20, 58]. We view existing program behavior as an expression of the domain-specific expertise of the original programmers. An automatic repair process can thus adapt existing techniques that analyze dynamic invariants over program data [22] to precisely reason about existing semantics, particularly as they relate to the location and nature of an error and its potential repair, encouraging both correctness and scalability. An initial automatic repair prototype tool that leverages portions of these insights and uses genetic programming has shown promise in preliminary work [25, 83], successfully repairing errors spanning 4 different vulnerability types in 11 legacy C programs totaling over 62,000 lines of code in under three minutes each, on average.

The expected main contributions of the proposed dissertation are:

1. **Automatic Repair:** A tool, which we call GenProg, ("Generic Program Repair"), that uses search strategies to automatically repair a varied set of existing errors in legacy software.
2. **Fault and Fix Localization:** Two new algorithms for automatically localizing a defect and its likely candidate repairs.
3. **Repair Templates:** An algorithm for mining templates of likely repairs as well as techniques to use such templates in the context of a search for a repair.
4. **Objective Functions:** A more precise objective function for use in search for program repair.

The rest of this proposal is organized as follows. Section 2 presents background and related work. Section 3 motivates and then outlines the proposed research in detail. Section 4 describes metrics and experimental strategies that will be used to evaluate the proposed research. Section 5 presents an initial implementation and preliminary results. Section 6 provides a timeline for the proposed dissertation research, and Section 7 concludes.

2 Background and Related Work

Genetic Programming. Stochastic search algorithms, such as simulated annealing, hill climbing, random search, and genetic programming, are optimization algorithms that incorporate probabilistic elements. *Genetic programming* (GP) is a stochastic search method, inspired by biological evolution, that discovers computer programs tailored to a particular task [24, 44]. GP maintains a population of tree-based representations of different programs, referred to as *individuals*, *chromosomes*, or *variants*. The *fitness*, or desirability, of each variant, is evaluated via an external *objective function* (typically called a *fitness function* in this context). High-fitness individuals are *selected* to be copied into the next generation. Computational analogies to the biological processes of mutation and crossover introduce variations into the population. These *mutation operators* create a new generation, and the cycle repeats, until either a goal or a resource limit is reached.

Although GP has solved an impressive range of problems (see, for example, [38]), most program repair GP efforts to date have been plagued by scalability problems [32]. The space of programs that must be searched is usually infinite, the objective function must scale while providing sufficient search feedback, and GP operators often introduce irrelevant code into intermediate variants while searching for a repair. Some previous efforts to apply GP to program repair [6] relied on formal specifications to evaluate fitness; this approach is time-intensive and limited, as specifications are rare in practice, difficult to generate [22], and limited in the types of errors they can prevent. In general, previous attempts [6, 7, 8, 60] are limited to very small, hand-coded examples. We propose novel representation choices to overcome these scalability concerns.

Search-Based Software Engineering. Search-based software engineering (SBSE) [33] applies evolutionary and related search methods to software testing, especially test suite generation, selection, and minimization [54, 79, 80], and engineering [3, 11, 69]. Many GP innovations in SBSE involve new kinds of fitness functions, and there has been less emphasis on novel representations or operators, as there is in the proposed work. However, test suite generation and evaluation are complementary to the proposed research, and we specifically propose new objective functions to use in a GP algorithm for program repair.

Debugging and fault localization. Debugging techniques aim to produce information to help programmers understand and correct software errors [88]. *Fault localization* is a particular debugging technique that identifies smaller sets of code likely to be implicated in a particular error (e.g. [4, 49, 53]). Ultimately, however, repairs for unannotated programs must still be generated manually. Additionally, despite the fact that many errors in real programs are addressed by adding code [83], such tools typically cannot localize the "cause" of an error to a missing statement. In general, debugging research is complementary to our own: an automatic repair technique can use localization or static analyses coupled with trace minimization techniques [9, 15, 31] as starting points in determining where to affect a repair. We propose to adapt fault localization techniques to improve scalability and generality in automatic program repair.

Dynamic Program Invariants. Techniques for learning and analyzing invariants over dynamic program data values (such as "$x > 0$") have been used to mine specifications [22], localize faults [50], and select templates to repair N-variant systems [63]. These techniques typically monitor program executions such that each execution, either on an indicative workload or during the course of normal operation, produces both program output and a set of invariants that were true during that run. This data is analyzed over many runs to identify invariants that appear to correlate with an event of interest, such as program failure. The result of a fault localization analysis, for example, might produce a set of facts similar to: "the branch condition at line 5 in file `foo.c` is often true when the program fails." We propose to leverage such dynamic semantic information to guide a search for a repaired program.

Automatic Error Preemption and Repair. Research in automatic error preemption and repair has ranged from using formal specifications to detect and patch errors [18, 81]; to automatically inserting code to handle overflow attacks using specific monitors [73, 74] or exception handlers [70]; to emulating vulnerable code [51, 72], suspicious requests [71], or candidate, templated repairs [63]. Several other recent but less mature proposals in this area, e.g., [17] suggest that we can expect rapid progress over the next several years.

There are two major limitations to these otherwise promising techniques. First, they require *a priori* enumeration of vulnerability types and repair approaches, often via formal specifications. Second, they increase code size or run-time or both, to a degree ranging from 20% [70] to 150% [74], usually precluding their application to legacy systems. We seek to address both of these issues in the proposed dissertation research.

Repair Templates. Repair templates have shown promise in automatic error preemption and repair; several existing approaches preemptively enumerate templates of repairs that they select and instantiate at repair time [51, 63, 71] using heuristics. Orthogonally, techniques and type systems for code templates feature prominently in run-time code generation and dynamic compilation research, which use run-time information, such as run-time constants, to compile code on-the-fly to speed execution [64]. Two well-known approaches, 'C [21] and DyC [30], propose template types, type systems, and techniques for selecting and instantiating templates to support dynamic compilation. We propose to generalize these techniques to develop template systems that usefully contribute to an automatic error repair algorithm.

Test Suites. Researchers have developed techniques for automatically generating test suites with varying coverage goals statically [16, 26, 27], dynamically [43, 75] and using a combination of static and dynamic analyses [14, 19, 29, 52, 68]. *Prioritization* (e.g., [67]) and *reduction* (e.g., [34]) techniques have been proposed to deal with the high cost of running test suites. *Impact analysis* [45, 61, 62, 65] uses static and dynamic analyses to determine which tests might possibly be affected by a source code change. As with fault localization and debugging research, faults identified by a test suite can be used as input by an automatic repair technique. Reduction, prioritization, and impact analyses can be used to reduce the time taken by the fitness function in the proposed application of search strategies to program repair [23]. However, we leave such extensions as future work and do not address them as part of the proposed dissertation.

3 Proposed Research

Our goal is to address some portion of the problem posed by ubiquitous and expensive errors in real-world systems by developing techniques to automatically repair them. At a high level, we propose to develop GenProg, an algorithm that takes as input a defective program, a *negative test case* that tests for the error, and a set of *positive test cases* that tests for required functionality. GenProg uses *mutation operators* to mutate the original program and perform a guided search through the space of closely-related programs, seeking a variant that passes all of the positive test cases (e.g. implements required functionality) as well as the negative test case (e.g. avoids the error). The search is guided by an *objective function*, which measures the behavior of a variant relative to the goal (e.g., passing all test cases). The choices underlying these aspects of GenProg are research questions that we propose to investigate; we focus particularly on leveraging and analyzing existing program semantics on test cases to increase generality and scalability.

We first motivate the proposed research by examining issues that arise when performing a search for a repair of an error adapted from a real-world vulnerability (Section 3.1). We then describe four proposed research thrusts:

1. **Using Search For Program Repair**. We propose to develop a baseline approach that uses guided search techniques to automatically repair program errors (Section 3.2). The development of the initial prototype and algorithm was performed in conjunction with other researchers and represents a part, but not the entirety, of this proposed dissertation research. Preliminary results have been previously published [23, 25, 82, 83].
2. **Fault and Fix Localization**. We propose to investigate how to accurately localize faulty code and identify code likely to affect a fix in the context of an automatic repair technique (Section 3.3).
3. **Repair Templates.** We propose to investigate techniques for mining and synthesizing templates of likely repairs, and develop new mutation operators to use them when generating closely-related variants in the space of nearby programs (Section 3.4).
4. **Objective Functions.** We propose to use dynamic invariants over program data in conjunction with test cases to construct a more precise and effective objective function to guide a search for a repaired program (Section 3.5).

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

(a) Buggy webserver code snippet.

```
 5       if(line == "Request:")
 6         strncpy(req_method, line+12)
 7       if(line == "Content-Length:")
 8         len=atoi(line+16)
 9     }
10     if (req_method == "GET")
11       buff=DoGETRequest(sock, len);
12     if(req_method == "POST") {
13 +     if(len > 0) {
14       buff=calloc(len, sizeof(char));
15       rc=recv(sock, buff, len);
16       buff[len]='\0';
17 +    } else {
18 +      buff=null;
19 +  }
20     }
21     return buff;
22 }
```

(b) Patch to repair the webserver.

Figure 1: A buggy webserver implementation, and a repaired portion of the same program.

Research thrust 1 serves to establish the baseline feasibility of applying search to automatic error repair. If successful, research thrusts 2–4 will increase the generality and scalability of automated error repair, allowing GenProg to be applied to more programs and bugs, and thus reduce development costs in more instances. This section describes the proposed research but does not discuss evaluation; Section 4 provides metrics and experiments for evaluating success.

3.1 Motivating Example

This section uses an example defect to highlight three key concerns in using search for automatic repair: **localization**, **repair templates**, and **objective functions**.

Consider the webserver pseudo-code in section (a) of Figure 1, adapted from a remote exploitable heap buffer overflow vulnerability in the `nullhttpd` webserver.[1] `ProcessRequest` handles an incoming request based on data copied from the request header. The code that processes POST requests contains a defect: line 13 allocates a buffer to hold the request content without bounds-checking the specified content-length, obtained on line 8. An attacker can remotely gain control of the running webserver by specifying a negative `Content-Length` in a POST request and providing a malicious payload in the request body. For the purposes of repair, we write a negative test case that checks for this error by sending such a POST request, and then checking whether the webserver is still running properly. We also write a positive test case that checks for required functionality by requesting `index.html` from the webserver and comparing the response against expected output.[2]

GenProg searches closely related programs by randomly mutating the defective program until it finds a version that passes all test cases; it may delete statements from the program, and it may insert or swap statements chosen uniformly at random from elsewhere in the program. The version in column (b) of Figure 1 shows the result of inserting a bounds-check from elsewhere in the webserver source code (the `cgi_main` function, which must also handle POST requests) at lines 13–17 of `ProcessRequest`. This variant passes both the positive and negative test case and does not

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

[2] In practice, requirements would be encoded using multiple test cases; we omit others for brevity.

include spurious changes; it serves as the final repair.

Unfortunately, the search process leading to this repaired variant can take too long, because the size of the search space of nearby programs is too great; we now highlight three ways to improve search performance as well as improve the generality of the automatic repair technique.

Localization. As the webserver incorporates far more code than is shown in Figure 1, the search for a repair may waste time investigating changes to irrelevant code. To reduce the search space, we propose to use *fault localization* to focus the random modifications on code that might affect the bad behavior, ideally without affecting good behavior. For example, a simple technique that identifies which program statements are executed on the negative test case but not the positive one focuses random changes on statements implementing POST functionality, on lines 13–17. More precise fault localization techniques might provide additional feedback to the mutation operators, or more accurately localize data-only vulnerabilities.

Even when the fault is localized, the number of possible changes remains very large, as random mutations may add or substitute arbitrary code. To address this, we leverage our hypothesis that a program that makes a mistake in one location often handles the situation correctly in another [20]. The general intuition holds for `nullhttpd`: the `cgi_main` function, implemented elsewhere, correctly bounds-checks the `Content-Length` of POST requests, and sets `buff` to `null` if `len < 0`; this portion of code, when inserted at line 13 of the code shown in section (a) of Figure 1, produces the repaired variant in section (b). GenProg uses heuristics to guide a search for the appropriate piece of code to insert at the fault location. We call the problem of identifying candidate portions of existing code that are more likely to affect a repair *fix localization*, and consider it the dual of fault localization.

Repair Templates. The error in `ProcessRequest` highlights the intuition that it should be possible to learn templates for common defects. Bounds-checking is common in C, and forgetting to do so is the source of many errors. Accordingly, it may be possible to create a template repair that generalizes bounds-checking, and add this template to the pool of candidate changes considered during the search for a repair. Moreover, a system for creating and using templates may increase the utility of fix localization, described above. For example, the repair described above and shown in (b) of Figure 1 requires that the code inserted from `cgi-main` uses the same names for local variables as the code in `ProcessRequest`; a repair template that could be legally instantiated in a new context would lift that restriction.

The Objective Function. The objective function defines the search stopping condition and measures the distance between a variant and the goal. Although the larger goal is to avoid the error, an objective function that measures only whether a variant passes the negative test case may yield undesirable results. For example, a version of `ProcessRequest` that returns `null` on all inputs passes the negative test case, but also fails to process any requests at all. The objective function must consider both the presence of correct behavior and the absence of incorrect behavior.

```
   if(req_method == "POST") {
 +   if(len > 0) {
     buff=calloc(len, sizeof(char));
     rc=recv(socket, buff, len);
     buff[len]='\0';
 +  } else {
 +    buff=null;
 + }
   }
 +  assert(false);
   return buff;
}
```

Figure 2: A variant that is close to the repair.

Additionally, an objective function that measures success only by test cases passed may be imprecise. Consider the variant of `ProcessRequest` shown in Figure 2, which differs from the repaired variant by only one line (line 21); however, this line causes it to fail all tests. Measured by tests passed, this variant is far from a repair.

A more precise objective function would allow the variant in Figure 2 to be retained, to help intermediate search steps. We propose to examine run-time data values to obtain this precision. For example, an examination of intermediate program state of the variant in Figure 2 shows that its behavior on the positive test case is almost unchanged (except for crashing on line 21), while its behavior on the negative test case has improved (it crashes instead of giving the attacker control). We propose to add such dynamic information to the objective function.

This example highlights three concerns in the design and implementation of an automatic repair technique based on search, and the ways that our proposed research thrusts can leverage dynamic program semantics and test cases to establish correctness while scalably and generally repairing a security defect. We will now outline proposed research to address each of these concerns.

3.2 Research: Using Search for Program Repair

The goal of this research thrust is to establish the feasibility of using search techniques, specifically genetic programming, to automatically repair programs. We propose several novel representation choices to overcome the dominant challenges of correctness and scalability [32]. We hypothesize:

> A search algorithm that (1) operates at the statement level of an abstract syntax tree, (2) uses test cases to localize a fault and define variant fitness, and (3) draws candidate repairs from elsewhere in the program source can automatically, quickly, and correctly repair a variety of defects in C programs, including legacy software.

The research in this subsection was performed in conjunction with other researchers and is described in detail in [25, 82, 83]. It serves as an initial prototype and baseline.

Approach. The output of this research thrust is an initial prototype of GenProg that uses genetic programming to search for a program variant that maintains required functionality without demonstrating a given defect. GenProg reduces the search space to a manageable size in several ways. First, it uses the positive and negative test cases to localize mutation operators to a smaller, likely-faulty subset of the program. The baseline localization approach is simple set intersection [39], as demonstrated in the motivating example (Section 3.1). Second, the GP mutations function at the level of the program's *abstract syntax tree* (AST), instead of at a lower level, for example, at the byte-level [60]. This choice increases the granularity of the search, further reducing the search space size. Finally, GenProg draws candidate changes exclusively and uniformly from elsewhere in the entire program source. The baseline objective function for calculating variant fitness is a weighted sum of the number of test cases passed by an variant. Additionally, GP often introduces irrelevant changes on the way to a solution, a problem known as *code bloat*. We control code bloat by combining a tree-structured differencing algorithm [2] with delta debugging [87], to efficiently find a *1-minimal* subset of the differences between a repaired variant and the original, defective program.[3] This minimal subset is called be the *final repair*; it may be encoded as a source-level program patch.

3.3 Research: Fault and Fix Localization

We define *fault* and *fix localization* to be the tasks of selecting, respectively, (1) a subset of a program to subject to mutation and (2) portions of source code to use as candidate repairs. The chosen

[3]For reasons of space we elide the details of this procedure; see [83, Section 3.5] for further detail.

localization techniques affect the types of errors GenProg can handle, and thus its generality. For example, data-only attacks, such as SQL injection attacks, are poorly-localized by the baseline set intersection technique proposed in Section 3.2. Moreover, both the set of statements considered for mutation (defined by fault localization) and the set of statements the mutation operator may use to attempt a repair (defined by fix localization) contribute to search space size, influencing scalability.

Previous research has shown how to learn dynamic invariants over program data [22] and how to use such invariants to precisely localize faults [50], mine specifications [22], and dynamically repair N-variant systems [63]. Other work suggests that there are a number of lightweight features of course code, such as churn or complexity, that correlate with bug density [13, 56, 77]. In previous work [47, 46], we showed that a combination of such software quality features can usefully augment static specification mining techniques. We propose to adapt those results to the domain of program repair, and hypothesize:

> A combination of static code quality metrics and learned dynamic behaviors can identify (1) candidate locations for automatic repair and (2) regions of source code that represent likely automatic repair fix candidates.

Approach. The output of this research thrust will be two algorithms that take as input a defective program and its positive and negative test cases. As output, they will associate with each program segment a probability that expresses its faultiness and repair potential, respectively. We propose to perform localization by combining measurements (called *features*) of program source and dynamic behavior. We first enumerate potential features, and then potential methods for their combination.

We first propose to adapt previous work in statistical bug isolation [50] (SBI) to learn predicates over program data that correlate with a defect under repair. SBI produces a simple mapping from a measure of faultiness to source code locations, suggesting a direct but incomplete starting point. We further propose to identify regions of code that are likely to impact apparently predictive invariants, based on either dynamic behavior (observed impact of the code region on relevant data values) or static source code features (reference to implicated variables). If necessary, we may develop a metric for identifying semantically similar regions of program source code as defined by mined invariant behavior, or adapt itemset mining techniques to the same purpose [1]. Second, we optionally propose to collect lightweight source code quality metrics, as in [46, 47]. However, the feasibility of this approach may be limited by the availability of suitable benchmarks.

There are several candidate approaches for combining the features into a coherent localization model. It is possible that a small subset of the SBI statistics will be sufficient to dramatically improve both fault and fix localization for program repair; we will begin by investigating several of these smaller sets, if only as baselines. We have had success in previous work using machine learning to construct a model that relate features to a dependent variable [46, 47]; a similar approach may prove beneficial here. Time permitting, we may investigate the application of belief-propagation techniques to localization for program repair [84].

3.4 Research: Repair Templates

We propose to use type system-informed approaches, potentially adapted from the run-time code generation community [21, 30], to mine candidate repair templates from program source (potentially informed by the fix localization algorithm proposed above) or code repositories. We further propose to develop new mutation operators that use repair templates in the context of an automatic search for a repair. We conjecture that a small set of templates, in addition to other candidate repair sources, may positively affect scalability without a large increase in search space size. This proposed approach generalizes previous repair techniques requiring templates [51, 63, 71] by leveraging our insight that existing program behavior contains the seeds of repair, and avoiding a restrictive preemptive enumeration of allowable templates. We hypothesize:

> An algorithm that synthesizes templates from existing source code or from source code repositories, in conjunction with new operators that can use such templates in mutation, can improve either the scalability or generality of search-based automated repair.

Approach. The output of this research thrust is an algorithm for synthesizing templates of possible repairs from program sources and source control repositories, as well as new mutation operators that can select and instantiate such templates to affect a repair. At a high level, a *template* is a piece of code, potentially including a set of "holes" that may be filled by in-scope variables. We will develop a distance metric on code changes (e.g., from a source code repository) or source code pieces to enable the use of clustering techniques [41] to identify clusters of similar changes or code snippets and identify "paragon" changes that encompass entire clusters to serve as templates.

To identify portions of the paragon template to serve as "holes", we will adapt either the proposed distance metric or type system approaches from run-time code generation research [21, 30]. In the latter approach, a paragon may be a template that encompasses a least-upper-bound type between several code changes in one cluster; this may require inference of simple type annotations. We may also be able to adapt notions of small-step contextual operational semantics to formalize the meaning of such a template [86].

Further, we propose to modify the mutation operator such that it may select a repair template as a potential program change in addition to its standard source code options. We will investigate semantics-oblivious mutation operators as well as operators that consider whether a template may be instantiated semantically legally; the latter approach may further leverage the type annotation approaches from run-time code generation research.

3.5 Research: Objective Functions

The *objective function* guides the automatic repair search by approximating the distance between a variant and the goal, e.g. a version of the program that passes all test cases. An objective function is ideally *precise*, accurately distinguishing intermediate variants representing partial solutions. Consider, for example, a program that contains a race condition that can be repaired by adding two distinct function calls: a lock acquisition and a lock release. A variant that contains only the lock acquisition should ideally have a higher fitness than the original (defective) program, as it is closer to the repair. However, the baseline proposed in Section 3.2, which evaluates a variant by counting how many test cases it passes, will likely return a lower fitness for this variant (e.g., as the unbalanced locking call may introduce deadlocks, leading to failed positive test cases). In the limit, an imprecise objective function transforms a guided search into a random search of closely-related variants, increasing the search space and decreasing the likelihood of finding repairs containing more than one change, and thus generality. We believe that the baseline approach is imprecise because it loses important intermediate information about dynamic program behavior. We therefore propose to further leverage learned dynamic invariant approaches, as in Section 3.3, to increase objective function precision. We hypothesize:

> An objective function that compares dynamic invariant behavior of the original and intermediate program variants on test cases can provide a precise signal to a program

repair search and permit either the more rapid repair of any error or the efficient discovery of repairs containing more than one change.

Approach. The output of this research thrust is a new objective function that uses intermediate dynamic behavior to measure the distance between a variant and a repair. We propose to observe dynamic invariant behavior on both the original program and intermediate variants. We will collect features to describe the different behavior of dynamic invariants on a program and its intermediate invariants. For example, we may count which invariants that were highly predictive of failure in the original program are no longer observed on an intermediate variant. We will then use machine learning to construct a model that relates observed dynamic behavior to fitness. Machine learning requires known data for a learning phase to construct a model. To enable this learning phase, we will develop an "oracular" distance function that expresses the *actual* distance between a program variant and a repaired variant. We propose to adapt tree-structured differencing techniques [2], or use a profile of dynamic predicate behavior on fixed variants, as starting points for this oracular distance function.

We have published, with other researchers, a proof-of-concept in [23]. We showed that it is possible to construct, for a single program, a model relating variant behavior quantified by dynamic invariants to an initial oracular function based on tree-based textual differences between a variant and a repair. The proposed research will expand on these results by generalizing the model to all programs, seriously investigating the problem of an oracular objective function, and incorporating the resulting model in the program repair search.

4 Proposed Experiments

This section describes the experiments and metrics we propose to use to evaluate the research proposed in Section 3. Beyond the individual contributions, the proposed research ultimately combines to create a system for automated program repair. To be successful, GenProg must be:

- **Scalable.** Previous work in automatic repair systems impose intractable run-time or code-size overhead. GenProg must scale to software modules of realistic size, such as hundreds of thousands of lines of code.
- **General.** To apply to legacy systems, GenProg must not require special coding practices, program annotations, or formal specifications, which are rare in practice. Additionally, previous research often requires *a priori* enumeration of vulnerability or repair types. GenProg should repair as generic a set of defect types with as little *a priori* knowledge as possible.
- **Correct.** To gain acceptance, GenProg must produce software patches that repair a defect as completely as possible while sacrificing as little functionality as possible.

Section 4.1 therefore proposes experiments to evaluate the initial approach (Section 3.2) in light of these three goals; we will also use these strategies to evaluate the contribution of the subsequent research to the system. Subsequent subsections outline experiments to individually evaluate the localization, repair templates, and objective function research. Research involving machine learning will be verified using 10-fold cross validation [42] to detect overfitting of the training data.

4.1 Experiments: Using Search for Program Repair

This subsection outlines an experimental plan for both the initial approach to applying stochastic search to automatic program repair (Section 3.2) as well as the contribution of each subsequent research thrust to the goal of automatic, scalable, general, and correct program repair. Much of the work from this research thrust has already been published, and the strategies for evaluating scalability and generality described here follows those and allow for direct comparison [25, 82, 83].

There are two important metrics for **scalability**: size of repaired programs and time taken to repair. As wall-clock time is strongly related to test suite used to evaluate fitness, and as initial work [23] suggests that running time can be managed with test suite selection techniques [65, 79], we will prefer generally to measure time to repair by number of fitness evaluations required.

To evaluate **generality**, we will seek a benchmark set containing a variety of program and error types, with a focus on off-the-shelf, legacy code. We will also seek (and hand-craft if necessary) programs containing specific error types as necessary for particular research thrusts.

We propose to measure **correctness** both manually (by inspecting the generated patches and, when available, comparing them to a hand-crafted repair) and using automated techniques, when possible on given benchmarks. First, we propose to use black-box fuzzers, such as the **SPIKE** tool from `immunitysec.com`, to generate large numbers (e.g., 100,000) of held-out fuzz inputs. We will measure how many of these inputs the benchmarks "get wrong" after the repair, to approximate a patch's impact on correctness and required functionality; Microsoft requires that security-critical changes be subject to exactly this type of test [37]. Second, we propose to use an off-the-shelf tool to fuzz exploit input, to evaluate whether the repairs truly repair the vulnerability, or if they simply represent fragile memorizations of the buggy input. Finally, we will measure benchmark performance in wall-clock time before and after repair, using workloads and held out tests.

4.2 Experiments: Fault and Fix Localization

We propose to use metrics from the debugging community, specifically the *score* metric [39], to evaluate the fault localization algorithm independently of its impact on program repair. Where possible, we will use related work in fault localization techniques as baselines. We will also evaluate the returned fault and fix candidates by comparing them manually to the "correct" answer (as defined by a human- or machine-generated fix, when available), similar to the approach of Liblit *et al.* [49, 50]. Additionally, we can quantify the instantiation of a given localization technique by summing the fault or fix probabilities associated with each statement in the program. We hypothesize that this sum approximates search space size and is related to algorithm success; we will investigate this hypothesis and, if it appears to hold, report this quantification for the new localization techniques. Finally, to evaluate the algorithms' effects on generality, we will seek out or hand craft difficult-to-localize errors, specifically data-only attacks such as SQL injection attacks.

4.3 Experiments: Repair Templates

To evaluate our new template mining and synthesis algorithms as well as templates' effects on time taken to repair, we will seek programs for which centralized version control information is available, such as applications on SourceForge, or from the IBM Jazz project, and especially benchmarks that contain errors repaired elsewhere in the source code or history (e.g., an error caused by a change from 32- bit to 64- bit operations that is not propagated to all relevant source locations). The proposed technique admits experimentation with mining algorithm parameters, particularly the definition of change or code similarity, how similar changes must be to justify forming a template, and how many similar changes are required to form a template. We propose to evaluate the new mutation operators that use repair templates in terms of their effect on scalability and, for example, by measuring the proportion of generated intermediate variants that are semantically valid.

4.4 Experiments: Objective Functions

Our ultimate goal is to create a precise objective function that can distinguish variants of a defective program that contain partial repairs. We will first focus on finding or creating benchmark programs that require more than one change to repair. We will therefore evaluate objective function precision by measuring the *fitness distance correlation* [40] between a new fitness function and an oracular distance function, following our previously-established methodology [23]. We will experiment with different oracular distance functions in both the training and testing phases of model generation and evaluation to identify an optimal solution.

5 Preliminary Results

This section describes and evaluates the initial implementation of GenProg described in Section 3.2 and published in [25, 82, 83]. We outline implementation, design, and representation choices in Section 5.1. We describe the experimental setup in Section 5.2. Section 5.3 presents experimental results on 11 benchmark C programs totaling 63 kLOC containing a total of 4 different error types.

5.1 Prototype Implementation

We use the CIL toolkit [57] to construct and manipulate abstract syntax trees (ASTs) of C programs; a CIL AST may be serialized as source code and passed to a compiler. The localization and GP operations function at the *statement level* of a a CIL AST; this includes all assignments, function calls, conditionals, blocks, and looping constructs. Program variants therefore consist of a pair of an AST and a *weighted path* through that AST.

The *weighted path* represents the application of a given localization approach to a particular error: each statement on the path is considered for mutation with probability equal to its weight, and only statements on the weighted path may be mutated. The prototype instruments the input program to record unique IDs for each statement as it is executed on the positive and negative test cases. Any statement visited exclusively by the negative test case is always considered for mutation; a statement that is executed by both a positive and a negative test case may also be considered, but with a lower (tunable) probability. Candidate repair statements are selected uniformly at random from anywhere else in the AST.

To evaluate variant fitness, GenProg compiles its AST to an executable program and then computes a weighted sum of all test cases passed by the executable. A variant that does not compile receives a fitness of zero. The search terminates when a variant is found that passes all test cases.

GenProg selects high-fitness individuals to create a new mating pool from the current generation; individuals in the mating pool are mutated at random and are the parents in the crossover operation. To perform selection, GenProg discards individuals with fitness 0 and then selects, based on fitness, up to half of the generation to retain. Two GP operators, mutation and crossover, create new program variants from the mating pool produced by the selection step:

- **Mutation.** Each variant in the mating pool is subject to mutation; each location on a variant's weighted path is considered for mutation with probability proportional to its path weight. The mutation operator consists of either a deletion (the entire statement is deleted), an insertion (another statement is inserted after it), or a swap with another statement; we choose from these three options with uniform random probability. In the case of an insertion or swap, a second statement is chosen uniformly at random from anywhere in the program; its new path weight is equal to the weight at the location to which it is being moved.

- **Crossover.** Each variant in the mating pool is crossed over no more than once with the original parent program; this type of crossover is sometimes referred to as *crossing back*. Only statements along the weighted paths are crossed over. GenProg randomly chooses a cutoff point along the paths and swap all statements after the cutoff point to create two new variants.

To control code bloat, the post-processing minimization step compares the variant containing the initial repair to the original program to generate a list of tree-based edits [2]. It then uses delta-debugging [87] to reduce this list of edits to a minimal repair.

5.2 Experimental Setup

Figure 3 shows our C benchmark programs, which include `unix` utilities taken from Milner *et al.*'s work on fuzz testing [55] and programs taken from public vulnerability reports; see [25] for references, where available. We used a single negative test case that elicits the fault for each benchmark. For the fuzz testing programs, we selected the first fuzz input that evinced a fault; for the others, we constructed test cases based on the vulnerability report. To select a small number of positive test cases for each program, we either used non-crashing fuzz inputs; or manually created simple test cases with a focus on testing a relevant subset of the benchmark's functionality.

Program	Lines	Description
`gcd`	22	example
`zune`	28	example [12]
`uniq utx 4.3`	1146	text processing
`look utx 4.3`	1169	dictionary lookup
`look svr 4.0 1.1`	1363	dictionary lookup
`units svr 4.0 1.1`	1504	metric conversion
`deroff utx 4.3`	2236	document processing
`nullhttpd 0.5.0`	5575	webserver
`indent 1.9.1`	9906	code text processing
`flex 2.5.4a`	18775	lexer generator
`atris 1.0.6`	21553	graphical tetris game
Total	63277	

Figure 3: Benchmarks used in our experiments, with program size in lines of code (LOC).

Our experiments were conducted on a quad-core 3 GHz machine. Beyond memoizing results based on the pretty-printed abstract syntax tree to avoid recalculating fitness on functionally equivalent variants, the prototype tool is not optimized. We report results for one set of global parameters that seemed to work well; see [83, Section 4.1] for details. We stop a trial if an initial repair is discovered. We performed 100 random trials for each program, reporting the fraction of successes and the time to find the initial repair. The initial repair is minimized to find the final repair. Minimization is deterministic and takes fewer seconds and fitness evaluations than the initial repair process.

5.3 Experimental Results

Figure 4 summarizes the repair results on our benchmarks. The '|Path|' column shows the *weighted path length*, the sum of the probabilities along the weighted path, roughly estimating the complexity of the search space. The 'Initial Repair' heading reports timing information for the genetic programming phase. The 'Time' column reports the wall-clock average time required for a trial that

Program	Fault	|Path|	Time	Fitness	Success	Initial	Final
`gcd`	infinite loop	1.3	149 s	41.0	54%	21	2
`zune`	infinite loop	2.9	42 s	203.5	72%	11	3
`uniq`	segmentation fault	81.5	32 s	9.5	100%	24	4
`look-u`	segmentation fault	213.0	42 s	11.1	99%	24	11
`look-s`	infinite loop	32.4	51 s	8.5	100%	21	3
`units`	segmentation fault	2159.7	107 s	55.7	7%	23	4
`deroff`	segmentation fault	251.4	129 s	21.6	97%	61	3
`nullhttpd`	remote heap overflow	768.5	502 s	79.1	36%	71	5
`indent`	infinite loop	1435.9	533 s	95.6	7%	221	2
`flex`	segmentation fault	3836.6	233 s	33.4	5%	52	3
`atris`	local stack overflow	34.0	69 s	13.2	82%	19	3
Average		801.56	171.7 s	52.0	59.9%	55.4	3.9

Figure 4: Experimental results on 63 kLOC. We report averages for 100 random trials. The '|Path|' column gives the weighted path length, which approximates search space size. 'Success' reports the percentage of random trials that resulted in a repair. 'Time' gives the average wall-clock time, and 'Fitness' the average number of fitness evaluations for a successful trial; neither includes minimization time. 'Initial' and 'Final' report the average Unix $\widehat{\text{diff}}$ size for both the initial and the minimized repair.

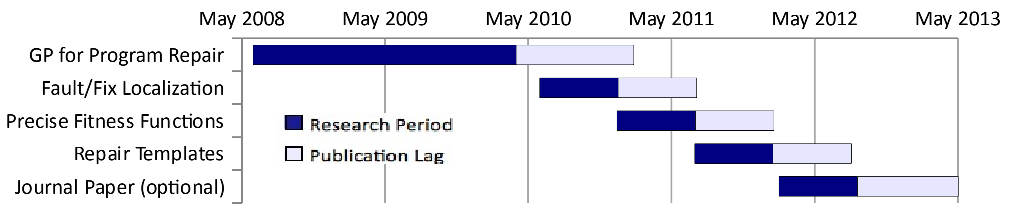

Figure 5: Proposed work schedule.

produced a primary repair; on average, less than 3 minutes are required to find the initial repair. The 'Fitness' column lists the average number of fitness evaluations performed during a successful trial. The 'Success' column gives the fraction of trials that were successful. On average, 60% of the trials produced a repair, although most of the benchmarks either succeeded very frequently or very rarely. The 'Size' column gives the size of both the initial repair and the final repair, in lines of code in the reified patch. The final minimized patch is quite manageable, averaging 4 lines of code.

This experiment demonstrates that the initial proposed repair approach (Section 3.2), that applies GP to automatic program repair is effective and promising. GenProg has successfully repaired several different error types in existing programs, in a reasonable amount of time.

6 Schedule

Figure 5 outlines the research schedule for the proposed dissertation. We propose to finish the proposed research and dissertation in three years, to graduate in May, 2013. We have completed the research described in Section 3.2 to establish the viability of using genetic programming for program repair [25, 82, 83]. A journal article on the subject is under revision with *Transactions on Software Engineering*. We have begun initial work on localization (Section 3.3) and expect to finish by January 2011. We have performed very preliminary research into the use of dynamic program invariants for precise fitness functions [23].[4] We have not begun work on the repair templates research (Section 3.4). The schedule includes considerable slack for projects not included in this proposal, including collaborative research on safety-critical medical equipment software with Professor John Knight; the completion of a publication based on a previous internship at Microsoft Research; and new research ideas that arise from the proposed research. We optionally intend to prepare an additional journal article based on the proposed dissertation research. We leave open the possibility of an additional research internship.

We have previously targeted venues such as *International Conference on Software Engineering (ICSE)*, *Genetic and Evolutionary Computation Conference (GECCO)*, and *Transactions on Software Engineering (TSE)*. We propose to target similar venues for the remaining work, as well as a top-tier programming languages venue like *Programming Language Design and Implementation (PLDI)* or *Principles of Programming Languages (POPL)*.

7 Conclusion

We argue that defects in deployed software are numerous and economically detrimental, and that the current practice of requiring programmers to address all such defects by hand is no longer sustainable. Accordingly, we propose to research new automatic techniques for repairing errors in legacy programs. Automated repair techniques suffer from the twin challenges of generality and scalability. We propose to leverage two insights to address these challenges: first, that test cases provide useful and sufficient insight into program behavior, and second, that existing program behavior contains the seeds of a program repair.

We propose four research thrusts. First, we propose a set of novel representation choices to effectively apply **search techniques to automatic program repair** to automatically find a variant of a defective program that does not contain the error but still implements required functionality. Our initial work has shown that this approach is both promising and effective [25, 82, 83]. Second, we propose to combine analyses of dynamic program semantics with static measures of code quality to develop **novel techniques for fault and fix localization**. This research is necessary for scalability because the chosen localization technique directly influences the search space size. Localization additionally affects generality: without better localization techniques, data-only attacks like cross-site scripting cannot feasibly be repaired. Third, we propose to generalize previous automatic repair research while further taking advantage of existing program behavior by developing techniques to **mine and synthesize repair templates**. Templates allow the repair of programs for which the fix must be found in a previous version or similar project, and the adaptation of more complex fixes from elsewhere in the program source. Finally, we propose to use test cases and existing research in dynamic invariant analysis to develop **precise and effective objective functions** for program repair. A precise objective function is necessary for the repair of many multi-step repairs, and directly impacts search efficiency, and thus scalability. We will evaluate each of these research thrusts on its own merits, as well as in the context of a tool for automatic defect repair, that we call GenProg, ("Generic Program Repair").

While we do not expect to develop a tool that can repair all defects in all programs, we do hope to gain insight into the nature of defective program behavior, and to develop tools and techniques to ease the ever-growing burden of error repair in software practice.

[4]The relevant paper included orthogonal work on test suite reduction and selection which we do not claim in this dissertation; we mention it for completeness.

References

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *International Conference on Management of Data*, pages 207–216, New York, NY, USA, 1993. ACM.

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

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

[4] L. Albertsson and P. S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workload. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 191, 2000.

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

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

[7] A. Arcuri, D. R. White, J. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *International Conference on Simulated Evolution And Learning*, pages 61–70, 2008.

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

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

[10] M.-C. Ballou. Improving software quality to drive business agility. White paper, International Data Corporation, `http://www.coverity.com/library/pdf/IDC_Improving_Software_Quality_June_2008.pdf`, June 2008.

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

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

[13] R. P. L. Buse and W. Weimer. A metric for software readability. In *International Symposium on Software Testing and Analysis*, pages 121–130, 2008.

[14] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating System Design and Implementation*, pages 209–224, 2008.

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

[16] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[17] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *International Conference on Automated Software Engineering*, 2009.

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

[19] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis*, pages 151–162, New York, NY, USA, 2007. ACM.

[20] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[21] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Principles of Programming Languages*, pages 131–144, 1996.

[22] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[23] E. Fast, C. Le Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolution Computation Conference*, pages 965–972, 2010.

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

[25] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference*, pages 947–954, 2009.

[26] P. Godefroid. Model checking for programming languages using verisoft. In *Principles of Programming Languages*, pages 174–186, New York, NY, USA, 1997. ACM.

[27] P. Godefroid. Compositional dynamic test generation. In *Principles of Programming Languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[28] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[29] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*. Internet Society, 2008.

[30] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.*, 248(1-2):147–199, 2000.

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

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

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

[34] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[35] K. J. Higgins. Cross-site scripting: attackers' new favorite flaw. Technical report, `http://www.darkreading.com/document.asp?doc_id=103774&WT.svl=news1_1`, Sept. 2006.

[36] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.

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

[38] 36 human-competitive results produced by genetic programming. `http://www.genetic-programming.com/humancompetitive.html`, Downloaded Aug. 17, 2008.

[39] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[40] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *International Conference on Genetic Algorithms*, pages 184–192, 1995.

[41] L. Kaufman and P. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Interscience, New York, 1990.

[42] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[43] B. Korel. Automated software test data generation. *IEEE Trans. Software Eng.*, 16(8):870–879, 1990.

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

[45] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *International Conference on Software Engineering*, pages 308–318, 2003.

[46] C. Le Goues and W. Weimer. Specification mining with few false positives. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 292–306. Springer, 2009.

[47] C. Le Goues and W. Weimer. Measuring code quality to improve specification mining. *IEEE Trans. Software Engineering (to appear)*, 2010.

[48] K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In K. Koskimies, editor, *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305, Apr. 1998.

[49] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, pages 141–154, 2003.

[50] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26, 2005.

[51] M. E. Locasto, A. Stavrou, G. F. Cretu, and A. D. Keromytis. From STEM to SEAD: speculative execution for automated defense. In *USENIX Annual Technical Conference*, pages 1–14, 2007.

[52] R. Majumdar and K. Sen. Hybrid concolic testing. In *International Conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[53] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *International Symposium on Foundations of Software Engineering*, pages 63–72, 2004.

[54] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *Transactions on Software Engineering*, 27(12):1085–1110, 2001.

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

[56] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Empirical Software Engineering and Measurement*, pages 364–373, 2007.

[57] 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.

[58] G. C. Necula and W. Weimer. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[59] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, NIST, May 2002.

[60] M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In *Genetic and Evolutionary Computation Conference*, pages 1043–1050. ACM, 2009.

[61] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. *SIGSOFT Softw. Eng. Notes*, 28(5):128–137, 2003.

[62] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *International Conference on Software Engineering*, pages 491–500, 2004.

[63] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles*, pages 87–102, October 2009.

[64] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 109–121. ACM Press, 1997.

[65] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not.*, 39(10):432–448, 2004.

[66] R. Richardson. FBI/CSI computer crime and security survey. Technical report, `http://www.gocsi.com/forms/csi_survey.jhtml`, 2008.

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

[68] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *European Software Engineering Conference/International Symposium on Foundations of Software Engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

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

[70] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Information Security*, pages 1–15, 2005.

[71] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[72] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS*, pages 37–48, 2009.

[73] A. Smirnov and T.-C. Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium*, 2005.

[74] A. Smirnov, R. Lin, and T.-C. Chiueh. Pasan: Automatic patch and signature generation for buffer overflow attacks. In *SSI*, 2006.

[75] M. L. Soffa, A. P. Mathur, and N. Gupta. Generating test data for branch coverage. In *Automated Software Engineering*, page 219, 2000.

[76] E. A. Strunk, X. Yin, and J. C. Knight. Echo: a practical approach to formal verification. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 44–53, 2005.

[77] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[78] Symantec. Internet security threat report. In *`http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf`*, Sept. 2006.

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

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

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

[82] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM Research Highlight*, 53(5):109–116, May 2010.

[83] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[84] Y. Weiss and W. T. Freeman. On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs. *IEEE Transactions on Information Theory*, 47(2):736–744, 2001.

[85] L. Williamson. IBM Rational software analyzer: Beyond source code. In *Rational Software Developer Conference*. `http://www-07.ibm.com/in/events/rsdc2008/presentation2.html`, June 2008.

[86] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

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

[88] A. Zeller. Automated debugging: Are we close? *IEEE Computer*, 34(11):26–31, 2001.