AFRL-RI-RS-TR-2012-225

GENERIC AND AUTOMATED RUNTIME PROGRAM REPAIR

UNIVERSITY OF VIRGINIA

SEPTEMBER 2012

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2012-225 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /

PATRICK M. HURLEY
Work Unit Manager

/ S /

WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
SEPTEMBER 2012	FINAL TECHNICAL REPORT	NOV 2010 – MAY 2012

4. TITLE AND SUBTITLE

GENERIC AND AUTOMATED RUNTIME PROGRAM REPAIR

5a. CONTRACT NUMBER
FA8750-11-2-0039

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Westley Weimer and Stephanie Forrest

5d. PROJECT NUMBER
G2AG

5e. TASK NUMBER
P0

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Virginia
Department of Computer Science
85 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740

University of New Mexico
Department of Computer Science
Mail Stop: MSC1 1130
1 University of New Mexico
Albuquerque, NM 87131-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2012-225

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the authors and should not be construed as an Official U.S. Government position, policy or decision, unless so designated by other documentation.

14. ABSTRACT
This document is the final technical report for Award FA8750-11-2-0039, "Generic and Automated Runtime Program Repair." The work carried out under this and other awards aimed to reduce costs associated with software maintenance by improving GenProg, a technique for the automated repair of program defects. This report discusses technical improvements to GenProg and its evaluation on over 100 defects in over 5 million lines of code involving over 10,000 test cases. In addition, this report details algorithms developed to automatically infer important program specifications: nonlinear invariants between program variables, invariants involving arrays, and invariants describing the use of programming interfaces. Finally, this report elaborates on techniques for scaling Gen- Prog to resource-constrained environments, such as embedded systems. Executable binaries and assembly-language programs for the x86 and ARM architectures are specifically targeted. Advances allow program repairs to be carried out using 15% of the memory and 5% of the disk space of pre-award work, and concrete examples of repairs using Nokia N900 smartphones are described. Overall the work partially supported by this award combines empirical and theoretical advances to significantly extend the reach of automated program repair.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**PATRICK M. HURLEY**
U	U	U	UU	40	19b. TELEPONE NUMBER (*Include area code)* **N/A**

Contents

List of Figures

List of Tables

1. Summary

A pressing challenge over the next decade is to produce and maintain software and hardware systems with fewer defects and more resilience to attack. Defects and vulnerabilities are reported so rapidly that programs routinely ship with known bugs, and even security bugs take 28 days, on average, to fix [1]. Software maintenance accounts for over $70 billion per year and is focused on repairing defects [2].

The work partially supported by this award aimed to reduce the time and effort gaps between finding and fixing software defects by improving a technique to automatically repair program bugs. The work improved a method for repairing defects in legacy applications through evolutionary computation and program analysis [3]. In this approach, no special coding practices are required. Instead, once a bug has been discovered, evolutionary algorithms evolve program variants until one is found that avoids the defect in question while retaining required functionality. Standard test cases are used to represent the fault and to encode program requirements. The improvement focused on two areas:

1. We investigated techniques for powerful and efficient **specification inference**. While automated program repair does not require formal specifications, they can reduce repair times, increase repair success, and improve confidence in finals repairs. Our primary research thrust in this area developed the first technique for discovering nonlinear polynomial and linear array invariants and was published in the International Conference on Software Engineering [4]. In addition, we developed a technique for discovering common software interface usage patterns, another form of specification. In a separately-funded human study involving over 150 participants, 82% of our specifications were found to be at least as good as human-written instances and 94% were strictly preferred to previous tool-assisted approaches (also published in the International Conference on Software Engineering [5]).

2. We extended previous work to **repair programs in many languages** and paradigms, including **embedded software**. Demonstrating applicability to other languages broadens the utility of automated program repair. We have repaired defects in programs written in five functional languages (LISP, Haskell, Ocaml, Clean, Scheme), one stack language (Forth), six object-oriented languages (Ada, Basic, C++, C#, Eiffel, Sather), and three imperative languages (Algol, BCPL, Fortran) beyond C. In addition, we targeted assembly language and ELF binaries for both ARM and x86 processors. Using new program representations and fault localization approaches, we observed a decrease of 68% in memory and 95% in disk space requirements, allowing program repair to scale to resource-constrained environments, as well as a 62% decrease in repair time compared to source-level approaches. This work has been published in the International Conference on Automated Software Engineering [6].

Overall, the work associated with this award is the first step toward making automated program repair, a promising technique for server-side C-program repair, viable for embedded devices, a large and important class of systems.

2. Introduction

Software bugs are ubiquitous, and fixing them remains a difficult, time-consuming, and manual process. Some reports place software maintenance, traditionally defined as any modification made on a system after its delivery, at 90% of the total cost of a typical software project [7, 8]. Modifying existing code, repairing defects, and otherwise evolving software are major parts of those costs [9]. The number of outstanding software defects typically exceeds the resources available to address them [10]. Mature software projects are forced to ship with both known and unknown bugs [11] because they lack the development resources to deal with every defect. For example, in 2005, one Mozilla developer claimed that, "everyday, almost 300 bugs appear ... far too much for only the Mozilla programmers to handle" [12, p. 363]. On the Mozilla project between 2002 and 2006, half of all fixed bugs took developers over 29 days each to fix [13]. This trend is particularly troubling in critical code: in 2006, it took 28 days on average for maintainers to develop fixes for security defects [1]. In a 2008 FBI survey of over 500 large firms, the average annual cost of computer security defects alone was $289,000 [14, p.16].

In light of this problem, many companies have begun offering *bug bounties* to outside developers, paying for candidate repairs. Well-known companies such as Mozilla[1] and Google[2] offer significant rewards for security fixes, with bounties raising to thousands of dollars in "bidding wars."[3]

Although security bugs command the highest prices, more wide-ranging bounties are available. Consider Tarsnap.com,[4] an online backup provider. Over a four-month period, Tarsnap paid $1,265 for fixes for issues ranging from cosmetic errors (e.g., typos in source code comments), to general software engineering mistakes (e.g., data corruption), to security vulnerabilities. Of the approximately 200 candidate patches submitted to claim various bounties, about 125 addressed spelling mistakes or style concerns, while about 75 addressed more serious issues, classified as "harmless" (63) or "minor" (11). One issue was classified as "major." Developers at Tarsnap confirmed corrections by manually evaluating all submitted patches. If we treat the 75 non-trivial repairs as true positives (38%) and the 125 trivial reports as overhead, Tarsnap paid an average of $17 for each non-trivial repair and received one about every 40 hours. Despite the facts that the bounty pays a small amount even for reports that do not result in a usable patch and that about 84% of all non-trivial submissions fixed "harmless" bugs, the final analysis was: "Worth the money? Every penny."[5]

Bug bounties suggest that the need for repairs is so pressing that companies are willing to pay for outsourced candidate patches even though repairs must be manually reviewed, most are rejected, and most accepted repairs are for low-priority bugs. These examples also suggest that

[1] `http://www.mozilla.org/security/bug-bounty.html` $3,000/bug

[2] `http://blog.chromium.org/2010/01/encouraging-more-chromium-security.html` $500/bug

[3] `http://www.computerworld.com/s/article/9179538/Google_calls_raises_Mozilla_s_bug_bounty_for_Chrome_flaws`

[4] `http://www.tarsnap.com/bugbounty.html`

[5] `http://www.daemonology.net/blog/2011-08-26-1265-dollars-of-tarsnap-bugs.html`

relevant success metrics for a repair scheme include the fraction of queries that produce code patches, monetary cost, and wall-clock time cost. In the work partially supported by this award we present an automated approach to program repair with a use case similar to that of the outsourced "bug bounty hunters." The method is powerful enough to fix over half of the defects it tackles, and we evaluate it using these and other metrics.

Until recently, most debugging approaches were manual and ad hoc. This work focused on a generic approach to automated software repair for security and software engineering defects, demonstrated most convincingly at the source level, but also applicable to other levels of the software stack. Our approach applies to off-the-shelf legacy applications without requiring formal specifications, program annotations or special coding practices. Once a program fault is discovered, we use evolutionary algorithms to search through program variants until one is found that both retains required functionality and avoids the defect. Standard test cases encode program requirements. After a successful repair has been discovered, it can be presented to developers for validation or applied to the program directly.

We have extended our technique's level of automation and its support for more realistic programs and defects. We wish to minimize human involvement in the repair loop, as well as targeting software on embedded devices, such as smartphones. Embedded failures have both civilian and military implications, ranging from lawsuits [15] to insurgents hacking Predator drone feeds [16]).

Ultimately, we have used our technique to repair over fifty-five distinct defects from off-the-shelf programs totaling over 5.1 million lines of code and involving over 10,000 test cases. In addition to infinite loops, segmentation faults, and bugs that produce incorrect output, we have repaired stack- and heap-based buffer overflows, non-overflow denial of service attacks, integer overflows, and format string vulnerabilities in at little as 36 minutes each, on average [3, Sec. V–A].

3. Methods, Assumptions, and Procedures

In this section we describe *GenProg*, an automated program repair method that searches for repairs to off-the-shelf programs (Section 2.1). We highlight the important algorithmic and representational changes since our pre-award preliminary work [17] that enable scalability to millions of lines of code, improve performance, and facilitate implementation on resource-constrained environments such as embedded systems or commodity cloud computing services (Section 2.1). We also describe our approach to dynamic invariant generation and program interface inference (Section 2.2). Finally, we summarize our work to apply GenProg to the domain of embedded software (Section 2.3).

3.1. Program Repair via Genetic Programming

GenProg uses *genetic programming (GP)* [18], an iterated stochastic search technique, to search for program repairs. The search space of possible repairs is infinitely large, and GenProg employs five strategies to render the search tractable: (1) coarse-grained, statement-level patches to

Input: Full fitness predicate FullFitness : $Patch \to \mathbb{B}$
Input: Sampled fitness SampleFit : $Patch \to \mathbb{R}$
Input: Mutation operator mutate : $Patch \to Patch$
Input: Crossover operator crossover : $Patch^2 \to Patch^2$
Input: Parameter $PopSize$
Output: Patch that passes FullFitness

```
1: let Pop ← map mutate over PopSize copies of ⟨ ⟩
2: repeat
3:    let parents ← tournSelect(Pop, Popsize, SampleFit)
4:    let offspr ← map crossover over parents, pairwise
5:    Pop ← map mutate over parents ∪ offspr
6: until ∃ candidate ∈ Pop. FullFitness(candidate)
7: return candidate
```

Figure 1: High-level pseudocode for GenProg repair technique main loop.

reduce search space size; (2) fault localization to focus edit locations; (3) existing code to provide the seed of new repairs; (4) fitness approximation to reduce required test suite evaluations; and (5) parallelism to obtain results faster.

High-level pseudocode for GenProg's main GP loop is shown in Figure 1. Fitness is measured as a weighted average of the positive (i.e., initially passing, encoding required functionality) and negative (i.e., initially failing, encoding a defect) test cases. The goal is to produce a candidate patch that causes the original program to pass all test cases. In this paper, each individual, or variant, is represented as a repair patch [19], stored as a sequence of AST edit operations parameterized by node numbers (e.g., Replace$(81, 44)\rangle$; see Section 3.1.1.).

Given a program and a test suite (i.e., positive and negative test cases), we localize the fault (Section 3.1.3.) and compute context-sensitive information to guide the search for repairs (Section 3.1.4.) based on program structure and test case coverage. The functions SampleFit and FullFitness evaluate variant fitness (Section 3.1.2.) by applying candidate patches to the original program to produce a modified program that is evaluated on test cases. The operators mutate and crossover are defined in Section 3.1.5. and Section 3.1.6. Both generate new patches to be tested.

The search begins by constructing and evaluating a population of random patches. Line 1 of Figure 1 initializes the population by independently mutating copies of the empty patch. Lines 2–6 correspond to one iteration or *generation* of the algorithm. On Line 3, *tournament selection* [20] selects from the incoming population, with replacement, parent individuals based on fitness. By analogy with genetic "crossover" events, parents are taken pairwise at random to exchange pieces of their representation; two parents produce two offspring (Section 3.1.6.). Each parent and each offspring is mutated once (Section 3.1.5.) and the result forms the incoming population for the next iteration. The GP loop terminates if a variant passes all test cases, or when resources are exhausted (i.e., too much time or too many generations elapse). We refer to one execution of the algorithm described in Figure 1 as a *trial*. Multiple trials are run in parallel, each initialized with a distinct random seed.

The rest of this section describes additional algorithmic details, with emphasis on the important improvements on our preliminary work, including: (1) a new patch-based representation (2) large-scale use of a sampling fitness function at the individual variant level, (3) fix localization to augment fault localization, (4) and novel mutation and crossover operators to dovetail with the patch representation.

3.1.1. Patch Representation

Over the course of this work we improved GenProg's representation for candidate repairs. Each variant is a *patch*, represented as sequence of edit operations (cf. [19]). In the original, pre-award algorithm, each individual was represented by its entire abstract syntax tree (AST) combined with a weighted execution path [17], which does not scale to memory-constrained environments such as embedded systems or commodity cloud computing settings. For example, for at least 35% of the defects considered in this evaluation, a population of 40–80 ASTs did not fit in the memory available for our main evaluation setup. In our dataset, half of all human-produced patches were 25 lines or less. Thus, two unrelated variants might differ by only 2×25 lines, with all other AST nodes in common. Representing individuals as patches avoids storing redundant copies of untouched lines. This formulation influences the mutation and crossover operators, discussed below.

3.1.2. Fitness Evaluation

To evaluate the fitness of a large space of candidate patches efficiently, we exploit the fact that GP performs well with noisy fitness functions [21]. The function SampleFit applies a candidate patch to the original program and evaluates the result on a random sample of the positive tests as well as all of the negative test cases. SampleFit chooses a different test suite sample each time it is called. FullFitness evaluates to true if the candidate patch, when applied to the original program, passes all of the test cases. For efficiency, only variants that maximize SampleFit are fully tested on the entire test suite. The final fitness of a variant is the weighted sum of the number of tests that are passed, where negative tests are weighted twice as heavily as the positive tests.

3.1.3. Fault Localization

GenProg focuses repair efforts on statements that are visited by the negative test cases, biased heavily towards those that are not also visited by positive test cases [22]. For a given program, defect, set of tests T, test evaluation function $Pass : T \to \mathbb{B}$, and set of statements visited when evaluating a test $Visited : T \to \mathcal{P}(Stmt)$, we define the fault localization function $faultloc : Stmt \to \mathbb{R}$ to be:

$$\mathit{faultloc}(s) = \begin{cases} 0 & \forall t \in T.\ s \notin \mathit{Visited}(t) \\ 1.0 & \forall t \in T.\ s \in \mathit{Visited}(t) \implies \neg \mathit{Pass}(t) \\ 0.1 & \text{otherwise} \end{cases}$$

That is, a statement never visited by any test case has zero weight, a statement visited only on a bug-inducing test case has high (1.0) weight, and statements covered by both bug-inducing and

normal tests have moderate (0.1) weights (this strategy follows pre-award work [17, Sec. 3.2]). On the defects considered here, the total weight of possible fault locations averages 110. Other fault localization schemes could potentially be plugged directly into GenProg [23].

3.1.4. Fix Localization

We introduce the term *fix localization* (or *fix space*) to refer to the *source* of insertion/replacement code, and explore ways to improve fix localization beyond blind random choice. As a start, we restrict inserted code to that which includes variables that are in-scope at the destination (so the result compiles) and that are visited by at least one test case (because we hypothesize that certain common behavior may be correct). For a given program and defect we define the function $\mathit{fixloc} : \mathit{Stmt} \rightarrow \mathcal{P}(\mathit{Stmt})$ as follows:

$$\mathit{fixloc}(d) = \left\{ s \;\middle|\; \begin{array}{l} \exists t \in T.\ s \in \mathit{Visited}(t) \wedge \\ \mathit{VarsUsed}(s) \subseteq \mathit{InScope}(d) \end{array} \right\}$$

The pre-award approach chose an AST node randomly from the entire program. As a result, an average of 32% of generated variants did not compile [17], usually due to type checking or scoping issues. For larger programs with long compilation times, this is a significant overhead. For the defects considered here, less than 10% of the variants failed to compile using the fix localization function just defined.

3.1.5. Mutation Operator

Pre-award work used three types of mutation: *delete*, *insert*, and *swap*. However, we found swap to be up to an order of magnitude less successful than the other two [24, Tab. 2]. We thus remove swap in favor of a new operator *replace* (equivalent to a delete followed by an insert to the same location). In a single mutation, a destination statement d is chosen from the fault localization space (randomly, by weight). With equiprobability GenProg either deletes d (i.e., replaces it with the empty block), inserts another source statement s before d (chosen randomly from $\mathit{fixloc}(d)$), or replaces d with another statement s (chosen randomly from $\mathit{fixloc}(d)$). As in previous work, inserted code is taken from elsewhere in the same program, but could also be adapted from learned specifications described correct program behavior. This decision reduces the search space size by leveraging the intuition that programs contain the seeds of their own repairs.

3.1.6. Crossover Operator

The *crossover* operator combines partial solutions, helping the search avoid local optima. We propose a new *patch subset* crossover operator, a variation of the well-known *uniform* crossover operator [25] tailored for the program repair domain. It takes as input two parents p and q represented as ordered lists of edits (Section 3.1.1.). The first (resp. second) offspring is created by appending p to q (resp. q to p) and then removing each element with independent probability one-half. This operator has the advantage of allowing parents that both include edits to similar ranges of the program (e.g., parent p inserts B after A and parent q inserts C after A) to pass any of those

edits along to their offspring. Previous uses of a one-point crossover operator on the fault localization space did not allow for such recombination (e.g., each offspring could only receive one edit to statement A).

3.2. Specification Inference

The study of program specifications or *invariants* (i.e., relations among variables that are guaranteed to hold at certain locations in a program) is a cornerstone of program analysis [26, 27, 28] and has been a major research area since the 1970s [29, 30, 31, 27, 32, 33]. Invariants can be identified using static or dynamic analysis. Static analysis is typically computationally expensive but more likely to provide provably sound results. Dynamic analysis is usually efficient, but its results are not guaranteed to be correct because the discovered properties may not generalize to all program traces. Nonetheless, dynamic invariant analysis is useful in practice because of its scalability and the help it can provide in program refactoring, documenting and debugging [34, 35, 36].

3.2.1. Nonlinear and Array Invariants

Nonlinear polynomial properties are essential to the success of many scientific, engineering and safety-critical applications. For example, Astrée [37, 38], a successful program analyzer used to verify the absence of run-time errors in Airbus avionic systems, implements a static analysis involving the ellipsoid abstract domain to represent and reason about a class of quadratic inequality invariants.[6] Nonlinear invariants have also been found useful for the analysis of hybrid systems [39, 40].

Arrays are a widely used data structure that is fundamental to many programs. For example, in C.A.R. Hoare's seminal 1971 paper on algorithm verification, *Proof of a program: FIND*, the overall goal is to prove an array invariant that lies at the heart of the correctness of quicksort [41, p.40]. Fixed-size arrays are also present in many systems programs, and proper analysis is often critical for security (e.g., buffer overruns). Finally, the ubiquity of arrays in general software engineering makes reasoning about arrays crucial for performance (e.g., for bounds check elimination [42]).

Daikon [34] is a well-known dynamic analysis system that detects invariants from program traces. However, Daikon supports only a limited form of linear relations among program variables and arrays. For example, Daikon cannot discover (i) that the location of the chosen pivot in binary search is $l + u - 1 \leq 2p \leq l + u$ as these inequalities involve three variables, (ii) that the gcd of x, y is $nx + my$ because this is a nonlinear polynomial, (iii) the equation $v = 2x + 3y + 4z + 5$ because it involves four variables, or (iv) the relation $A[i] = B[C[i]]$ because it is a nested array relation. It is thus difficult to fully capture and reason about the semantics of programs that can only be expressed in such forms of invariants with Daikon.

To address the issues outlined above and improve dynamic invariant detection, we combined mathematical techniques that have not been previously applied to the problem: equation solving, polyhedra, and SMT (Satisfiability Modulo Theories) solving. More specifically, we focused on

[6] The ellipsoid domain used by Astrée [38] to examine the Airbus system is expressed in the quadratic form $x^2 + axy + by^2 \leq k$, where $0 < b < 1$ and $a^2 < 4b$.

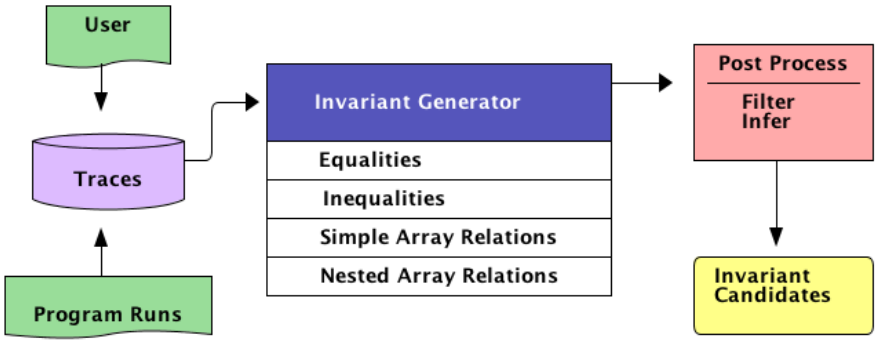


Figure 2: Automatic Generation of Dynamic Invariants. The generator finds different types of invariants from program traces. The post-processing step removes redundant and spurious invariants.

generating invariants expressed as nonlinear arithmetic relations among program variables and invariants on relations among complex data structures such as multi-dimensional arrays.

We developed a technique to find equalities among nonlinear *polynomials* of program variables using equation solving. We find nonlinear inequalities by constructing convex polyhedra. When additional inputs from the user are available, our approach can also deduce new inequalities from previously obtained equality relations.

We also developed a technique to find linear equalities among *arrays* by first finding equalities among array elements and then identifying the relations among array indices from the obtained equalities. We find nested array relations by performing reachability analysis. Our analysis has potentially high time complexity, and thus we encode the problem as a satisfiability problem, which can be efficiently solved with an SMT solver.

We view dynamic invariant detection as two *separate* subproblems: (i) fixing a priori candidate invariants over program variables and then (ii) ruling out invalidated candidates based on observed traces of program variables. We hypothesize that a key reason that previous approaches do not scale to nonlinear invariants or array invariants is that they enumerate candidates based on fixed templates (e.g., linear equations involving at most three variables). Such eager enumeration strategies do not scale to higher-degree polynomials or array invariants due to the large number of possible candidates. By contrast, our approach lazily explores the search space based on the structure of the trace data. It considers candidate invariants based on the traces available, rather than an eager enumeration. This insight, coupled with tools such as equation and SMT solvers, allows us to find human-relevant nonlinear and array invariants in nontrivial programs efficiently. We implemented a prototype tool, depicted in Figure 2, based on this approach.

3.2.2. Nonlinear Invariant Example

Invariants are typically placed at the entries and exits of functions corresponding to pre- and post-conditions and/or the heads of loops corresponding to loop invariants. Given a location l, the program is instrumented to trace the values of the variables in scope at l. The instrumented program is then run against a set of inputs to obtain the traces.

```
1  int cohendiv(int x, int y){
2    int q = 0; // quotient
3    int r = x; // remainder
4    while (r >= y) {
5      int a = 1;
6      int b = y;
7      while (r >= 2 * b) {
8        // Invariant Location
9        // Invs: b=ya, x=qy+r, r >= 2ya
10       a = 2 * a;
11       b = 2 * b;
12     }
13     r = r - b;
14     q = q + a;
15   }
16   return q;
17 }
```

Figure 3: Cohen's integer division algorithm.

The program in Figure 3 implements the well-known integer division algorithm by Cohen [43], which takes as input two integers x, y and returns the integer q as the quotient of x and y. We consider invariants at location l, the head of the inner while loop on line 9. There are six variables $\{a, b, q, r, x, y\}$ in scope at l. Table 1 consists of five sets of values representing traces obtained from the variables at l for inputs $\{x = 15, y = 2\}$ and $\{x = 4, y = 1\}$.

Table 1: Traces of the Cohen program on inputs $\{x = 15, y = 2\}$ and $\{x = 4, y = 1\}$.

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
4	1	1	1	0	4
4	1	2	2	0	4

We want to obtain the polynomial invariants over the variables $\{a, b, q, r, x, y\}$ based on such traces. The documented invariants $\{b = ya, x = qy + r, r \geq 2ya\}$, which cannot be identified with current dynamic invariant methods, describe precisely the semantics of the inner while loop

in Cohen's algorithm.[7] The full details of our algorithms have been published in the International Conference on Software Engineering [4].

3.2.3. Program Interface Invariants

Professional software developers spend most of their time trying to understand code [44, 45]. Maintaining and evolving high-quality documentation is crucial to help developers understand and modify code [46, 47]. In reports by and studies of developers, use examples related to an application program interface (API) have been found to be a key learning resource [48, 49, 50, 51, 52, 53]. That is, documenting how to *use* an API is preferable to simply documenting the function of each of its components.

One study found that the greatest obstacle to learning an API in practice is "insufficient or inadequate examples." [54] We developed an algorithm that automatically generates API usage examples. Given a data-type and software corpus (i.e., a library of programs that make use of the data-type), our approach extracts abstract use-models for the data-type and renders them in a form suitable for use by humans as documentation or suitable for mechanical use as specifications.

The state of the art in automated support for usage examples is known as *code search*. Typically, the problem is phrased as one of ranking concrete code snippets on criteria such as "representativeness" and "conciseness." In 2009, Zhong *et al.* described a technique called MAPO for mining and recommending example code snippets [55]. More recently, Kim *et al.* presented a tool called eXoaDocs which also finds and ranks code examples for the purpose of supplementing JavaDoc embedded examples [56]. Such examples can be useful, but they are very different from human-written examples. Mined examples often contain extraneous statements, even when slicing is employed. In addition, they often lack the context required to explicate the material they present. In general, mined examples are long, complex, and difficult to understand and use. Good human-written examples, on the other hand, often present only the information needed to understand the API and are free of superfluous context. Human written documentation has two important disadvantages, however: it requires a significant human effort to create, and is thus often not created; and it may not be representative of, or up-to-date with, actual use.

We developed a technique for automatically synthesizing human-readable API usage examples which are well-typed and representative. We adapted techniques from specification mining [57] to model API uses as graphs describing method call sequences, annotated with control flow information. We use data-flow analysis to extract important details about each use beyond the sequence of method calls, such as how the type was initialized and how return values are used. Our approach then abstracts concrete uses into high-level examples. Because a single data-type may have multiple common use scenarios, we use clustering to discover and coalesce related usage patterns before expressing them as documentation.

Our generated examples display a number of important advantages over both state-of-the-art code search and human written examples, both of which we compare to in a human study. Unlike mined examples, our generated examples contain only the program statements needed to demon-

[7] The invariant $x = qy + r$ asserts that the dividend x equals to the divisor y times the quotient q plus the remainder r.

strate the target behavior. Where concrete examples can be needlessly specific, our examples adopt the most common types and names for identifiers. Unlike human-written examples, our examples are, by construction, well-formed syntactically and well-typed. Where previous approaches to code ranking adopted simple heuristics based on length and a simple use count (e.g., [58, 59]), our abstract examples are structured and generated with a robust and well-defined notion of representativeness. Because our approach is fully automatic, the examples are also cheap to construct and can be always up-to-date. Additionally, their well-formedness properties make them ideal automated tasks like for code completion [60] or program repair.

3.2.4. Program Inference Example

In modern software development, API documentation tools such as JAVADOC have become increasingly prevalent, and variants exist for most languages (e.g., PYTHONDOC, OCAMLDOC, etc.). One of the principles of JAVADOC is "including examples for developers" [61]. Not all examples are created equal, however. Features such as conciseness, representativeness, well-chosen variable names, correct control flow, and abstraction all relate to documentation quality.

Consider Java's `BufferedReader` class, which provides a buffering wrapper around a lower-level, non-buffered stream. The human-written usage example included in the official Java Development Kit, version 6 [62] is:

```
1 BufferedReader in =
2  new BufferedReader(new FileReader("foo.in"));
```

While this example has the merit of being concise, it shows only how to create a `BufferedReader`, not how to use one. By contrast, our algorithm produces:

```
1 FileReader f; //initialized previously
2 BufferedReader br = new BufferedReader(f);
3 while(br.ready()) {
4    String line = br.readLine();
5    //do something with line
6 }
7 br.close();
```

This exemplifies one common usage pattern for a `BufferedReader`: repeatedly calling its `readLine` method while it remains `ready`. The variable names `br` and `f` were selected from among the most common human choices for `BufferedReader` and `FileReader`, and were synthesized together here: no single usage example need exist that uses both of those names in tandem. In addition, the example also demonstrates the importance of control flow: `readLine` is called repeatedly, but only after checking `ready`. Finally, the `//initialized previously` and `//do something with line` comments indicate points where different human developers would write different code and highlight the most direct places for a developer to adapt this code example into an existing setting.

In practice, there is more than one way to use a `BufferedReader`. Our algorithm can produce a ranked list of examples based on clusters of representative human usages. The second example we produce is:

```
1 InputStreamReader i; //initialized previously
2 BufferedReader reader = new BufferedReader(i);
3 String s;
4 while ((s = reader.readLine()) != null)
5    //do something with s
6 }
7 reader.close();
```

This second example shows that other concrete argument types can be used to create a `BufferedReader` (e.g., a `InputStreamReader` can be used as well as a `FileReader`). In addition, it shows that there is a different usage pattern that involves always calling `readLine` but then checking the return value against `null` (rather than calling `ready`). Both of the examples produced by our algorithm are well-formed and introduce commonly-named, well-typed temporaries for function arguments and return values.

It is also possible to use code search and slicing techniques to produce API examples. Such a tool from Kim *et al.* [56] produces an output consisting of more than 14 lines on the same `BufferedReader` query (not shown).

Because they lack information related program semantics, slicing based approaches have trouble distinguishing between relevant an irrelevant details in an example. In the output of Kim *et al.*'s tool, a `BufferedReader` is initialized with `System.in`. To a new user, it may be difficult to tell if this argument is necessary, or as in this case, coincidental. Variable names are also often too specific to the example: `String acl_in = br.readLine()` (Kim *et al.*'s tool) is less descriptive than `String line = br. readLine()` for the general case. Furthermore, sliced examples do not type-check out of context and include many irrelevant statements.

Our approach produces high-quality usage example documentation automatically. The details of our algorithm are published in the International Conference on Software Engineering [5].

3.3. Program Repair for Embedded Software

Few automated repair techniques apply to mobile or embedded systems, instead targeting desktop client software such as Firefox [63, 64], server software such as MySQL [63] or webservers [21], or design-by-contract Eiffel programs [65]. This is unfortunate because embedded failures have both civilian and military implications, ranging from faulty-firmware lawsuits [15] to insurgents hacking Predator drone feeds [16]. One example of a wide-reaching embedded defect was the "Zune bug", in which 30GB Microsoft Zune Media Players froze up on the last day of a leap year [66]. In addition, previous repair techniques that apply to binaries [64] or assembly language [67] have uniformly targeted Intel x86, despite "the widespread dominant use of the ARM processor in mobile and embedded systems" [68].

In this award we extended our automated program repair work to run directly on compiled assembly files (*ASM*) and linked ELF executables (*ELF*). To do so, we introduced a stochastic method of fault localization appropriate for these lower-level representations. This extension removes the requirement for source code availability and the need for compilation and linking as part of the search process. It also allows finer-grained (sub-statement) mutations.

In large programs, it is reasonable to assume that most parts of the program are *not* related to a given bug [69]. Accurate fault localization is thus critical for our approach and is an important factor in running time. Common fault localization methods assume that the bug is likely to be associated with code executed when operating on the bug-inducing input.

Previous program repair approaches thus record entire sequences or paths [17] of executed statements using various weighting factors [69]. One key challenge for scaling automated program repair to embedded systems was to obtain information that is accurate enough to guide automated repairs but inexpensive enough to be gathered on resource-constrained devices.

Collection of execution information at the ASM and ELF levels requires a new fault localization technique. Many traditional code profilers (e.g., `gcov`) are language specific and rely on the insertion of assembly instrumentation. For example, previous attempts to repair the `flex` benchmark [17] were C-language specific and stored sequential ordering information from about 443,399 raw statement visits, with the instrumentation increasing the CPU time required by a factor of $100.4\times$. Direct extensions such as deterministic sampling of the program counter (e.g., using `ptrace`) did not perform adequately in our preliminary work.

To address these constraints, we developed a modified stochastic sampling approach to fault localization. We first sample the program counter (PC) across multiple executions of the program. These PC values are then mapped to bytes in the `.text` section of ELF files and to specific instructions in ASM files. Stochastic sampling only approximates control flow, and the memory addresses reported are often insufficient to guide the repair process (e.g., gaps, elided periodic behavior, etc.). To overcome these limitations, we apply a 1-D Gaussian convolution to the sampled addresses with a radius of 3 assembly statements. This has the effect of increasing the number of instructions implicated by each sample. Gaussian convolution is an accepted method of smoothing data to reduce detail and noise [70].

Our assembly-level representation is a linear sequence of assembly instructions (e.g., as produced by `gcc -S`). Candidate repairs are generated through swapping, copying, duplicating or deleting assembly instructions. Pseudo-operations and assembly directives (e.g., `.section .rodata`) are retained but not modified. To reduce search space size, each assembly instruction is treated atomically, and its operands are not mutated individually. This representation is source language and architecture agnostic.

The Executable and Linkable Format (ELF) is a popular representation for object and executable files on modern Unix operating systems. No access to source code or to any intermediate stage of the build process are required to manipulate ELF files. The instructions in the `.text` section are treated as a linear array of bytes grouped into whole assembly instructions (i.e., opcodes with their arguments), possibly of variable width, which are modified by the ELF-level mutation operations.

As in the ASM representation, each instruction is treated as atomic and opaque. When the size of the instruction array changes, many parts of the ELF file must be updated. These include (1) the headers of the `.text` section; (2) sections subsequent in memory; and (3) all ELF data structures with pointers to targets which have moved (e.g., `.dynsym`, `.reloc`, etc.). Unfortunately, there are often hard-coded memory addresses included as literals in the `.text` section. Since there is no general way to distinguish an integer literal from an address literal, such references are left

Table 2: Subject C programs, test suites and historical defects. Tests were taken from the most recent version available in May, 2011.

Program	LOC	Tests	Defects	Description
`fbc`	97,000	773	3	legacy coding
`gmp`	145,000	146	2	precision math
`gzip`	491,000	12	5	data compression
`libtiff`	77,000	78	24	image processing
`lighttpd`	62,000	295	9	web server
`php`	1,046,000	8,471	44	web programming
`python`	407,000	355	11	general coding
`wireshark`	2,814,000	63	7	packet analyzer
total	**5,139,000**	**10,193**	**105**	

unchanged.

The details of our approach for applying automated program repair to embedded systems have been published in the International Conference on Automated Software Engineering [6].

4. Results and Discussion

In this section we present empirical results related to automated program repair, focusing on advances made during the period of this award (Section 3.1); specification inferences (Section 3.2); and the application of program repair to embedded systems (Section 3.3).

4.1. Program Repair

Our goal was to select an unbiased set of programs and defects that can run in our experimental framework and is indicative of "real-world usage." We required that *subject programs* contain sufficient C source code, a version control system, a test suite of reasonable size, and a set of suitable subject defects. We only used programs that could run without modification under cloud computing virtualization, which limited us to programs amenable to such environments. We required that *subject defects* be reproducible and important. We searched systematically through the program's source history, looking for revisions that caused the program to pass test cases that it failed in a previous revision. Such a scenario corresponds to a human-written repair for the bug corresponding to the failing test case. This approach succeeds even in projects without explicit bug-test links, and it ensures that benchmark bugs are important enough to merit a human fix and to affect the program's test suite.

Table 3: Repair results: 55 of the 105 defects (52%) were repaired successfully. The total cost of generating the results in this table was $403.

Program	Defects Repaired	Cost per Non-Repair Hours	Cost per Non-Repair US$	Cost Per Repair Hours	Cost Per Repair US$
`fbc`	1 / 3	8.52	5.56	6.52	4.08
`gmp`	1 / 2	9.93	6.61	1.60	0.44
`gzip`	1 / 5	5.11	3.04	1.41	0.30
`libtiff`	17 / 24	7.81	5.04	1.05	0.04
`lighttpd`	5 / 9	10.79	7.25	1.34	0.25
`php`	28 / 44	13.00	8.80	1.84	0.62
`python`	1 / 11	13.00	8.80	1.22	0.16
`wireshark`	1 / 7	13.00	8.80	1.23	0.17
total	**55 / 105**	**11.22h**		**1.60h**	

4.1.1. Program Repair: Experiment

Table 2 summarizes the programs used in our experiments. We selected these benchmarks by first defining predicates for acceptability, and then examining various program repositories to identify first, acceptable candidate programs that passed the predicates; and second, all reproducible bugs within those programs identified by searching backwards from the checkout date (late May, 2011). Defects are defined as test case failures fixed by developers in previous versions. The next subsection formalizes the procedure in more detail.

We ran 10 GenProg *trials* in parallel for each bug. We chose $PopSize = 40$ and a maximum of 10 generations for consistency with previous work [17, Sec. 4.1]. Each individual was mutated exactly once each generation, crossover is performed once on each set of parents, and 50% of the population is retained (with mutation) on each generation (known as elitism). Each trial was terminated after 10 generations, 12 hours, or when another search found a repair, whichever came first. SampleFit returns 10% of the test suite for all benchmarks.

We used Amazon's EC2 cloud computing infrastructure for the experiments. Each trial was given a "high-cpu medium (c1.medium) instance" with two cores and 1.7 GB of memory.[8] Simplifying a few details, the virtualization can be purchased as *spot instances* at $0.074 per hour but with a one hour start time lag, or as *on-demand instances* at $0.184 per hour. These August–September 2011 prices summarize CPU, storage and I/O charges.[9]

[8] `http://aws.amazon.com/ec2/instance-types/`
[9] `http://aws.amazon.com/ec2/pricing/`

4.1.2. Program Repair: Results

Table 3 reports results for 105 defects in 5.1 MLOC from 8 subject programs. GenProg successfully repaired 55 of the defects (52%), including at least one defect for each subject program. Successful results are reported under the "Cost per Repair" columns. The remaining 50 are reported under the "Non-Repair"s columns. "Hours" columns report the wall-clock time between the submission of the repair request and the response, including cloud-computing spot instance delays. "US$" columns reports the total cost of cloud-computing CPU time and I/O. The 50 "Non-Repairs" met time or generation limits before a repair was discovered. We report costs in terms of monetary cost and wall clock time from the start of the request to the final result, recalling that the process terminates as soon as one parallel search finds a repair. Results are reported for cloud computing spot instances, and thus include a one-hour start lag but lower CPU-hour costs.

For example, consider the repaired `fbc` defect, where one of the ten parallel searches found a repair after 6.52 wall-clock hours. This corresponds to 5.52 hours of cloud computing CPU time per instance. The total cost for the entire bug repair effort to repair that defect is thus $10 \times 5.52 \text{ hours} \times \$0.074/\text{hour} = \$4.08$.

The 55 successful repairs return a result in 1.6 hours each, on average. The 50 unsuccessful repairs required 11.22 hours each, on average. Unsuccessful repairs that reach the generation limit (as in the first five benchmarks) take less than 12+1 hours. The total cost for all 105 attempted repairs is $403, or $7.32 per successful run. These costs could be traded off in various ways. For example, an organization that valued speed over monetary cost could use on-demand cloud instances, reducing the average time per repair by 60 minutes to 36 minutes, but increasing the average cost per successful run from $7.32 to $18.30.

Table 3 does not include time to minimize a repair, an optional, deterministic post-processing step. This step is a small fraction of the overall cost [17].

We view the successful repair of 55 of 105 defects from programs totaling 5.1 million lines of code as a very strong result for the power of automated program repair. Similarly, we view an average per-repair monetary cost of $7.32 as a strong efficiency result. Further details and additional evaluations for this work are available [3]. In Section 3.2 we present results in specification inference that hold out the promise of increasing the fraction of defects that can be repaired and in Section 3.3 we present results applying these techniques to resource-constrained embedded systems.

4.2. Specification Inference

We evaluated our dynamic invariant generation prototype on programs taken from a test suite which we call NLA (nonlinear arithmetic) and an implementation of AES encryption. We evaluated our approach to inferring program interfaces on 1.3 million lines of Java code.

4.2.1. Dynamic Invariant Generation: Experiment

Our first benchmark, the NLA test suite, consists of 24 programs from various sources collected by Rodríguez-Carbonell and Kapur [71, 72]. These programs implement classic arithmetic algorithms

that are widely used in programming, such as mult, div, pow, mod, sqrt, gcd, lcm. The programs are relatively small, about 20 lines of C code each. However, they implement nontrivial mathematical algorithms and are often used to benchmark static analysis methods. Importantly, the complexity of our method depends on the size of the traces, the number of variables of interest, and the type of relations among program variables but *not* the size of the program per se. Among the 24 programs from NLA, there are 35 documented nonlinear invariants: 33 are equations and 2 are inequalities.

The second benchmark, AES, is an annotated AES implementation from Yin *et al.* [73]. It exemplifies a real-world security-critical application and contains nontrivial array invariants. To show that functions in the AES implementation conform to the formal AES specification, the implementation authors inspected and documented the invariants of each function and then fully verified the result using SPARK Ada and PVS. The annotated invariants represent the manual effort required to fully functionally verify an AES implementation using axiomatic semantics. The AES implementation contains 868 lines of Ada code organized into 25 functions containing 30 invariants: 8 simple array relations, 7 nested array relations, 2 linear equations, and 13 other relations.

Our test programs come with documented invariants at various locations such as loop heads and function exits. For evaluation purpose, we manually instrumented the source code of the programs to trace values of all variables in the scope at each program location containing a known invariant. Our goal is to find invariants at those locations automatically and compare them to the human-documented invariants.

The instrumented programs were run against a set of randomly selected inputs. The number of obtained traces is different across programs and program locations. For example, locations inside loops may be visited many times while function exits may be visited rarely.

4.2.2. Dynamic Invariant Generation: Results

Table 4 lists our experimental results on 24 programs from NLA, averaged over 20 runs. The *Vars* column reports the number of distinct variables in that program's invariants, *Deg* reports the highest polynomial degree in those invariants, *Invs* reports the number of invariants found by our approach and the total number of documented invariants, and *T* reports the average time in seconds to discover the invariants, including the time to refine the results.

We found all 35 documented nonlinear invariants from the NLA test suite. In most cases, the results matched the documented invariants exactly as written. Occasionally, we achieved results that are mathematically equivalent to the documented invariants. For example, the sqrt1 program has two documented equalities $2a + 1 = t$ and $(a + 1)^2 = s$, our results give $2a + 1 = t$ and $t^2 + 2t + 1 = 4s$, which is equivalent to $(a + 1)^2 = s$ by substituting t with $2a + 1$. We note that current dynamic analysis approaches cannot find any of these nonlinear relations.

Table 5 lists our experimental results on 25 functions from AES, averaged over 20 runs. The *Arrs* column reports the number of distinct arrays in that program's invariants, *Dim* reports the highest dimension of the arrays in those invariants, *Inv Types* reports the types of invariant: *S*imple, *N*ested, and *O*thers. $N(d)$ specifies that the nesting depth is d. The *driver* functions are composed from other functions in this table.

Table 4: Experimental results for dynamic invariant generation on 24 programs from NLA.

Program	Desc	Inv Types	Vars	Deg	Invs	T (s)
divbin	div	eq	5	2	1/1	0.5
cohendiv	div	eq, ieq	6	2	2/2	1.3
mannadiv	int div	eq	5	2	1/1	0.3
hard	int div	eq	6	2	1/1	0.9
sqrt1	sqr	eq, ieq	4	2	2/2	0.7
dijkstra	sqr	eq	5	2	1/1	0.5
freire1	sqr	eq	3	2	1/1	0.2
freire2	cubic root	eq	4	3	2/2	3.2
cohencube	cube	eq	5	3	3/3	12.6
euclidex1	gcd	eq	10	2	3/3	6.5
euclidex2	gcd	eq	8	2	2/2	2.5
euclidex3	gcd	eq	12	2	4/4	10.1
lcm1	gcd, lcm	eq	6	2	1/1	0.5
lcm2	gcd, lcm	eq	6	2	1/1	0.6
prodbin	product	eq	5	2	1/1	0.3
prod4br	product	eq	6	3	1/1	8.1
fermat1	divisor	eq	5	2	1/1	0.8
fermat2	divisor	eq	5	2	1/1	0.4
knuth	divisor	eq	8	3	1/1	71.5
geo2	geo series	eq	4	2	1/1	0.2
geo3	geo series	eq	5	3	1/1	3.1
ps2	pow sum	eq	3	2	1/1	0.1
ps3	pow sum	eq	3	3	1/1	0.3
ps4	pow sum	eq	3	4	1/1	0.8
24 programs		**2 types**			**35/35**	**126.0s**

We found all 17 relations that are expressible in our considered forms. The other 13 invariants do not fall into categories described above and are left for future work. These can be grouped into three categories: Others$_{1-3}$. Others$_1$ includes nested array invariants such as $A[i] = 4B[6C[\dots]]$. We current do not handle such nested invariants when the elements of A are not exactly nested in B. Others$_2$ includes array invariants such as $A[i] = B[\dots]$ where $i = \{0, 4, 8, 12, \dots\}$ and $A[i'] = B[\dots]$ where $i' = \{1, 2, 3, 5, 6, 7, 9, 10, 11 \dots\}$. We require that generated relations such as $A[i] = B[\dots]$ hold for all i. Others$_3$ includes array invariants involving functions whose inputs are arrays, such as $f([1, 2])$. We only consider functions with scalar inputs such as $g(7, 8)$. We note that existing dynamic analysis methods cannot find these array relations either.

The manual annotation of AES with sufficient invariants to admit machine-checked full formal verification was a significant undertaking involving hours of tool-assisted manual effort [73, 74].

Table 5: Experimental results for dynamic invariant generation on 25 functions from AES.

Function	Desc	Inv Types	Arrs	Dim	Invs	T (s)
multWord	mult	N(4)	7	2	1/1	3.6
xor2Word	xor	N(1)	4	2	1/1	0.1
xor3Word	xor	N(1)	5	3	1/1	0.1
subWord	subs	N(1)	3	1	1/1	0.4
rotWord	shift	S	2	1	1/1	0.5
block2State	convert	S	2	2	1/1	2.0
state2Block	convert	S	2	2	1/1	11.7
subBytes	subs	N(1)	3	2	1/1	0.6
invSubByte	subs	N(1)	3	2	1/1	3.8
shiftRows	shift	S	2	2	1/1	12.2
invShiftRow	shift	S	2	2	1/1	8.3
addKey	add	N(1)	4	2	1/1	0.6
mixCol	mult	O_3	4	2	0/1	-
invMixCol	mult	O_3	4	2	0/1	-
keySetEnc4	driver	S,O_2	2	2	1/2	4.5
keySetEnc6	driver	S,O_2	2	2	1/2	6.7
keySetEnc8	driver	S,O_2	2	2	1/2	10.6
keySetEnc	driver	O_3	4	1	0/1	-
keySetDec	driver	O_3	4	2	0/1	-
keySched1	driver	O_1	3	2	0/1	-
keySched2	driver	O_1	3	2	0/1	-
aesKeyEnc	driver	eq,O_3	7	2	1/2	0.1
aesKeyDec	driver	eq,O_3	7	2	1/2	0.1
aesEncrypt	driver	O_3	8	4	0/1	-
aesDecrypt	driver	O_3	8	4	0/1	-
25 functions		**6 types**			**17/30**	**65.9s**

Annotating pre- and postconditions and loop invariants has not been solved in general and is known to be a key bottleneck in approaches based on axiomatic semantics [75]. It is not surprising that our approach was unable to discover all relevant invariants; indeed, we view reducing the manual verification annotation burden by one-half as a strong result.

To summarize, we found all of the invariants under consideration: 100% of the documented nonlinear invariants in NLA and 17 out of 30 documented invariants in AES. The other 13 invariants are beyond the scope of this paper and left for future work. On average, it takes under five seconds to find the invariants for each program. To the best of our knowledge, no other dynamic invariant analysis approaches have analyzed the forms of invariants discussed in this paper. Additional experiments and information are available [4].

Table 6: Subject programs used to evaluate API inference.

Name	Version	Domain	kLOC
FindBugs	1.2.1	Code Analysis	154
FreeCol	0.7.2	Game	91
hsqldb	1.8.0	Database	128
iText	2.0.8	PDF utility	145
jEdit	4.2	Word Processing	123
jFreeChart	1.0.6	Data Presenting	170
tvBrowser	2.5.3	TV Guide	138
Weka	3.5.6	Machine Learning	402
XMLUnit	1.1	Unit Testing	10
total			1361

4.2.3. Program Interface Inference: Experiment

In this subsection we evaluate our program interface algorithm and prototype implementation with one qualitative metric: human acceptability. Running our prototype tool on 1,361k lines of code to produce example documentation for 35 classes took 73 minutes (about 2 minutes per class). About 95% of this time is spent filtering the corpus and enumerating paths.

Throughout our evaluation we compare the output of our tool to both human-written examples from the Java SDK and also the eXoaDoc tool of Kim *et al.* [56]. eXoaDoc works by leveraging an existing code search engine to find examples of a class. Kim *et al.* then employ slicing to extract "semantically relevant" lines. These examples are then clustered and ranked based on *Representativeness*, *Conciseness* and *Correctness* properties. eXoaDoc has been shown to increase productivity by as much as 67% in a small study.

Our dataset, shown in Table 6, consists of examples from all 35 classes from standard Java APIs for which we have one example of each of the three types. Because eXoaDocs are associated with methods rather than classes, we chose the top example for the most popular method (by static count of concrete uses in our benchmark set). For our tool, we chose the top (i.e., most representative) example for each class.

The goal of this study is to quantify the desirability of the output of our tool in comparison to both human-written examples (from the Java SDK) and the state-of-the-art tool eXoaDoc of Kim *et al.* [56]. The study involved 154 participants evaluating 35 pairs of API examples.[10]

For each of the 35 classes from the dataset, the participant is shown the name of the target class and is randomly shown two of the three documentation types (i.e., ours, eXoaDoc, and human-written). The participant is then required to make a preference evaluation by selecting one option from a five-element Likert scale: "Strong Preference" for A, "Some Preference" for A, "Neutral",

[10]The human study evaluating this algorithm was *not* funded by this award. Instead, it was funded by National Science Foundation award CCF 1116289, "SHF: Small: Synthesizing Human-Readable Documentation". The results are reproduced here as relevant to the evaluation of the overall specification inference work performed.

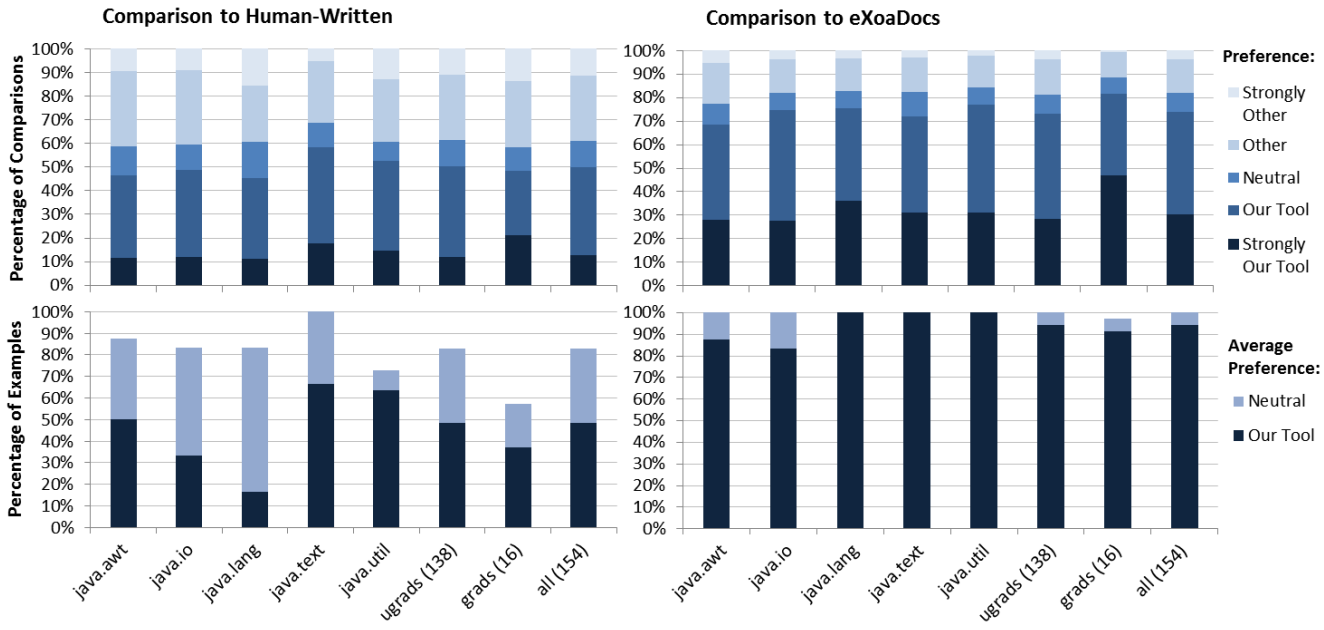


Figure 4: Aggregated results from our human study. The left charts show responses comparing our tool to human-written examples and the charts on the right compare our tool to the EXOADOC tool of Kim *et al.* [56]. The charts on the top categorize all comparisons (e.g., in 60% of comparisons, our tool output was judged at least as good as human-written examples). The charts on the bottom reflect the consensus opinion for each example (e.g., 82% of examples produced by our tool were judged to be at least as good as human-written on average). Examples are grouped by package (e.g., `java.X`) and by category of participants (138 ugrads, 16 grads).

"Some Preference" for B, "Strong Preference" for B. If desired, the participant may also choose to "Skip" the pair.

The study was advertised to students at The University Of Virginia enrolled in a class entitled *Software Development Methods*, which places a heavy focus on learning the Java language and its standard set of APIs. For comparison, 16 computer science graduate students also participated. 179 students participated in total, however, to help preserve data integrity, we removed from consideration the results from 25 undergraduate students who completed the study in less than five minutes. The average time to complete the study for the remaining 154 participants was 13 minutes.

Participation was voluntary and anonymous. The participants were instructed to "Pretend that [they] are a programmer or developer who needs to use, or understand how to use, the class." No additional guidance was given; participants formed their own opinions about each example.

4.2.4. Program Interface Inference: Results

In total, 154 participants compared 35 example pairs each, producing 5,390 distinct judgments. The aggregate results are presented in Figure 4. Overall, the output of our tool was judged at least as good as human written examples over 60% of the time and strictly better than EXOADOC

in about 75% of cases. For 82% of examples, on average either humans preferred our generated documentation to human-written examples or had no preference. For 94% of examples, the output of our tool was preferred to EXOADOCS.

Raw score distributions were very similar between graduate and undergraduate students. However, taken example by example, grad students had a 20% reduced preference for tool-generated examples when compared to human-written examples. Nonetheless, both grads and undergrads judged our generated documentation to be at least as good as gold-standard human-written for over half of the examples. Compared to EXOADOC, our tool was preferred in almost all cases by both groups.

Finally, we asked whether "a program that automatically generates examples like these would be useful?", and 81% agreed that it would be either "Useful" or "Very Useful." Further information and additional evaluations are available [5].

Taken together, the results in Section 3.2 suggest that we can infer previously-untouched classes of invariants, including half of those needed for formal verification, in one example, and that our inference results are as good as what humans would write over half of the time. This work provides significant confidence that specification inference is positioned to augment and inform automated program repair.

4.3. Program Repair for Embedded Software

This subsection presents results of an empirical evaluation of program repair at the ASM and ELF levels. The results show the following:

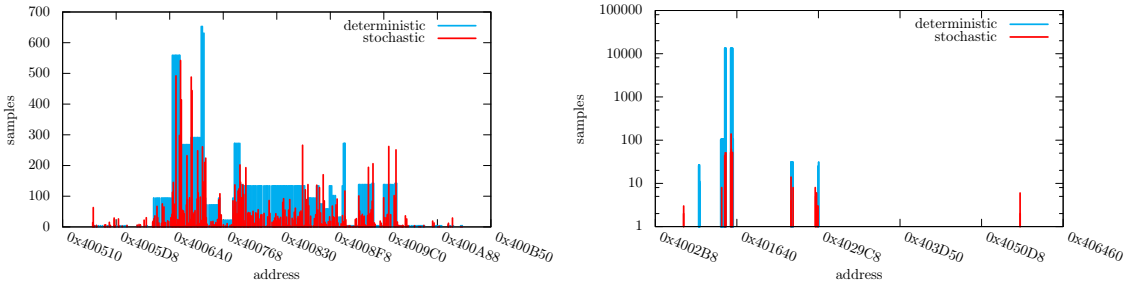
1. Stochastic fault localization closely approximates the deterministic approach. (Section 4.3.1.)
2. Repair success at the ASM and ELF representation levels is similar to that reported previously for ASTs. (Section 4.3.2.)
3. Our ASM and ELF representations, together with our stochastic fault localization method, have small resource footprints, suitable for running on mobile and embedded devices. (Section 4.3.3.)

Table 7 lists the benchmark defective programs evaluated in this paper. The benchmarks are generally taken from pre-award work on AST-level repair [17] for the purposes of direct comparison; **`merge`** sort was added to evaluate the stochastic fault localization algorithm on a full-coverage test suite. Each program comes equipped with a regression test suite, used to validate candidate repairs, and a special test case indicating a defect (bug). These programs have on average 3.69 more assembly instructions and 9.52 more ELF bytes than lines of source code. Sizes are given for each representation: Lines of code (LOC) in the original C source, LOC in the assembly files (x86, as produced by `gcc -S`), and size (in bytes) of the `.text` sections of the x86 ELF files.

We used the following genetic algorithm parameters: population size $\mathsf{popsize} = 1000$; maximum number of fitness evaluations before terminating a trial $\mathsf{evals} = 5000$, mutation rate $\mathsf{mut} = 1.0$ per individual per generation and crossover rate $\mathsf{cross} = 0.5$ crossovers per indivdiual per generation.

Table 7: Subject programs for ASM and ELF repair experiments, taken from Weimer *et al.* [17].

Program	C LOC	ASM LOC	ELF Bytes	Program Description	Defect
`atris`	9578	39153	131756	graphical tetris game	local stack buffer exploit
`ccrypt`	4249	15261	18716	encryption utility	segfault
`deroff`	1467	6330	17692	document processing	segfault
`flex`	8779	37119	73452	lexical analyzer generator	segfault
`indent`	5952	15462	49384	source code processing	infinite loop
`look-s`	205	516	1628	dictionary lookup	infinite loop
`look-u`	205	541	1784	dictionary lookup	infinite loop
`merge`	72	219	1384	merge sort	improper sorting of duplicate inputs
`s3`	594	767	1804	sendmail utility	buffer overflow
`uniq`	143	421	1288	duplicate text processing	segfault
`units`	496	1364	3196	metric conversion	segfault
`zune`	51	108	664	embedded media player	infinite loop
total	31791	117261	302748		



(a) `merge` (b) `deroff`

Figure 5: Fault localization in program address space. The stochastic results are shown in *light green*; they identify similar program regions as the previous deterministic approach, shown in *dark red*.

Experiments were run either on a machine with 2.2 GHz AMD Opteron processors and 120 GB of memory or on Nokia N900 smartphones, each of which features a 600 MHz ARM Cortex-A8 CPU and 256 MB of mobile DDR memory, as appropriate.

4.3.1. Embedded Software: Fault Localization Results

Our stochastic fault localization method samples the contents of the program counter and processes the results. We used `oprofile` [76], a system-wide profiler for Linux systems, to sample the PC. We then mapped these sampled memory addresses back to individual bytes in the code sections of ELF executables and to instructions in assembly files and process with Gaussian convolution.

We evaluated the resulting fault localization in the context of program repair in Section 4.3.2. and also explicitly compared the results of the stochastic and deterministic approaches. For this comparison we used `merge` sort, which is small and exhaustively tested (i.e., 100% statement and branch coverage), as well as `deroff`, which is larger with less-complete test coverage. The stochastic and deterministic traces taken from the failing test cases of both programs are shown overlaid in Figure 5.

Ten stochastic samples and one deterministic sample were taken for each program. We find very high correlations of 0.96 (merge) and 0.65 (deroff) between stochastic samples, indicating consistency across samples. We find high correlations of 0.61 (merge) and 0.38 (deroff) between the naïve stochastic (no convolution) and deterministic samples, which increase to 0.71 (merge) and 0.48 (deroff) after Gaussian convolution, indicating the post-processing step is important.

4.3.2. Embedded Software: Repair Success Results

Table 8 compares ASM and ELF to the AST representation used previously [17]. The "Memory" column reports the memory usage when performing repairs for each combination of program and representation. Our Nokia smartphones have 256+768 MB of RAM plus swap. In this embedded environment 8 of 12 programs can be repaired at the AST level compared to 9 at the ASM level and 10 at the ELF level. "Runtime" reports the average time per successful repair in seconds.

Table 8: Resource comparison of abstract syntax tree (AST) assembly source (ASM) and ELF binary (ELF) representations.

	Memory (MB)			Runtime (s)			% Success			Expected Fitness Evaluations		
Program	AST	ASM	ELF	AST	ASM	ELF	AST	ASM	ELF	AST	ASM	ELF
`atris`	2384*	2384*	496	22.87	†	385.63	83	0	5	27.44	†	48806.00
`ccrypt`	6437*	3338*	334	39.15	342.23	21.58	100	100	100	7.00	673.00	25.00
`deroff`	1907*	811	453	37.33	1366.61	292.88	100	98	100	48.00	50.00	454.00
`flex`	691	381	162	1948.84	1125.44	†	6	1	0	78340.50	496255.00	†
`indent`	3242*	1669*	572	3301.88	3852.47	†	4	41	0	62737.25	13517.48	†
`look-s`	420	62	29	747.59	353.81	6.00	100	100	100	41.00	71.00	3.00
`look-u`	430	52	62	12.68	6.38	3.66	100	100	100	90.00	16.00	19.00
`merge`	152	45	57	842.74	100.93	161.35	54	100	84	4456.85	621.00	1008.19
`s3`	152	76	43	14.43	23.46	28.02	100	96	50	4.00	4.00	95.00
`uniq`	358	72	72	105.18	3.46	7.18	100	100	100	8.00	46.00	8.00
`units`	572	162	95	1075.16	18778.70	501.54	91	13	51	930.23	57374.63	8538.47
`zune`	76	17	29	36.93	28.79	71.49	100	100	100	17.00	26.00	45.00
average	1402	756	200	323.47	2333.82	121.52	78.17	70.75	65.83	622.45	6542.40	1132.85

† Indicates that there were no successful repairs in 5000 fitness evaluations. Rows with † are excluded when calculating "Runtime" and "Expected Fitness Evaluations" averages. * Indicates memory requirements in excess of the 1024 available on the Nokia smartphones.

The "% Success" column reports the number of runs out of 100 that successfully found a repair within the first 5000 test suite evaluations disregarding memory limitations. Differences among the average percentage of successful repairs across representations are not large, with values of 65.83%, 70.75% and 78.17% for ELF ASM and AST respectively. "Expected Fitness Evaluations" counts the expected number of evaluations per repair (see Equation 1).

Some bugs are more amenable to repair at particular levels of representation. For example, `atris` and `units` are easily repaired at the AST level, `indent` is most easily repaired at the ASM level and `merge sort` is most easily repaired at the ASM and ELF levels. We discuss the `atris` and `merge` repairs in greater detail.

The `atris` repair involves deleting a call to `getenv`. At the AST level this requires a single deletion, while at the ASM level the deletion of three contiguous instructions is needed. There are no nearby jump instructions, making it impossible to remove these three instructions with a single mutation. The intermediate variants (with some subset of the three deleted) have 0 fitness. The ELF representation can repair this defect, and all five repairs found were unique, but each involved from 3 to 7 accumulated mutation operations.

The `merge` repair involves replacing an `if` statement with its `else` branch. At the AST level, this is most easily accomplished by swapping the `if` with its `else`, which is 1 of 4900 possible swap mutations. At the ELF level, deletion of a single comparison instruction (one of 218 possible deletions) suffices to repair the program.

The "Expected Fitness Evaluations" column reports the expected number of fitness evaluations per repair.

$$
\begin{aligned}
expected &= fit_s + (run_s - 1) \times fit_f \text{ where} \\
fit_s &= \text{average evaluations per successful run} \\
fit_f &= \text{average evaluations per failed run} \\
run_s &= \text{average runs per success}
\end{aligned}
\tag{1}
$$

Because repair time is dominated by fitness evaluation (including compilation at the AST level and linking at the ASM level) and that for all programs but `units` (an outlier in this regard) the expected number of evaluations is roughly equivalent between levels of representation, we conclude that, when repairs are possible, the repair process is as efficient at the ASM and ELF levels as at the AST level.

4.3.3. Embedded Software: Resource Requirements Results

We desire a repair algorithm that can run within the resource constraints of mobile and embedded devices. We consider three key constraints: CPU usage and runtime, memory requirements, and disk space requirements. Table 8 highlights results.

CPU Usage. Runtime costs associated with genetic algorithm bookkeeping (e.g., sorting variants by fitness, choosing random numbers, etc.) are typically dwarfed by the cost of evaluating fitness. For example, on an average run of `deroff`, bookkeeping accounted for only 13.5% of the runtime. The primary costs are computing fault localization information fitness evaluation (including time to compile and assemble variants, which varies by representation).

Our stochastic fault localization requires from 50 to 5000 runs of the original unmodified program. Importantly, the absolute running time determines the number of required executions, so slow programs require fewer executions and only quickly terminating programs require more than 50 executions. By contrast, the previous AST-level work requires compilation of an instrumented program with a $100\times$ slowdown per run, previous executable-level approaches introduce a $300\times$ slowdown to compute richer fault localization information [64, Sec 4.4], and `ptrace` full deterministic tracing incurs a $1200\times$ slowdown. Our fault localization approach is an order-of-magnitude faster than these previous approaches.

For post-localization repair runtime, ASM and ELF have lower fitness evaluation costs than AST. Compared to AST-level approaches, ASM does not require compilation and ELF does not require assembling or linking, both reducing runtime. However, for the same program, the search space for ASM and ELF is larger than the corresponding source-level search space (see size columns in Table 7). If the time to conduct a repair at the AST level is 1.0, on average ASM repairs take $7.22\times$ and ELF repairs take $0.38\times$. The ELF improvement is particularly striking, but both are positive results.

Memory. Memory utilization is especially important for mobile and embedded devices. Previous approaches report results on repairs conducted on server machines with 8 GB [64] to 16 GB of ram [17]. By contrast, the Nokia N900 smartphones we consider as indicative use cases have 256 MB Mobile DDR, an order of magnitude less.

Table 8 reports the memory used (in MB) for repairs at the AST, ASM and ELF representations. We note that ASM requires only about 53.91% of the memory of a source-based representation, while ELF is significantly smaller, requiring only 14.29% of the memory. We attribute the low requirements for ELF to the ELF parser we used, which only stores the `.text` section of ELF binaries in memory.

Disk space. Beyond the subject program and its test suite, disk usage was composed of two main elements: the evolutionary repair tool itself and the build suite of the program to be repaired. The size of these requirements varies greatly with the level of representation. For example, repairs at the ELF level do not require that the build tool-chain of the original program be supplied. This facilitates local repair of embedded programs which are often cross-compiled and cannot be built locally. We review the disk space requirements at all three levels.

AST requires the source code and build tool chain of the original program. Our baseline comparison, the pre-award GenProg AST-level evolutionary repair tool [17], takes 23 MB on disk (including the tool itself, the **`gcc`** compiler and header files, the **`gas`** assembler, and the **`ld`** linker).

ASM requires only the assembly code, assembler, and linker. This is a significantly lighter build requirement. Our ASM representation is currently incorporated into the AST repair framework [17] to ensure a controlled environment for comparison. It requires 12 MB on disk (including the tool itself, the **`gas`** assembler, and the **`ld`** linker).

ELF requires only a compiled executable. Like ASM, our prototype is a modification of the AST-level repair framework, replacing the source-code parser with an ELF parser. It requires only 1.10 MB on disk, an order of magnitude decrease compared to AST.

As one concrete example of the resource limitations of embedded devices, the Nokia N900 smartphone ships with 256 MB of NAND flash storage (holding the Maemo Linux kernel and bootloader, etc., with about 100 MB free), and a 32 GB eMMC store holding a 2GB `ext3` partition, 768 MB of swap, and about 27 GB of free space in a `vfat` partition. The `vfat` partition is unmounted and exported whenever a USB cable is attached to the device, making it unsuitable for a deployed system repair tool. Linux packages install to the NAND flash by default, quickly exhausting space. Repartitioning is possible but uncommon for casual users. Thus, even though the device claims 32 GB of storage, significantly less is available for a stable repair tool. These are all merely implementation details, but we claim that such exceptions and a desire to minimize the on-disk footprint are indicative of many mobile and embedded devices.

5. Conclusions

Software maintenance remains critical and expensive. The work partially supported by this award focused on improving GenProg, an automated program repair technique for reducing the costs associated with software defects. Over the period of this award, advances in GenProg's representations and algorithms (Section 2.1) led to a full-scale, systematic evaluation using 105 bugs in over 5 million lines of code involving over 10,000 test cases. Using commodity cloud-computing infrastructure, the approach was able to repair over half of the defects encountered for $8 each (Section 3.1). We worked to augment GenProg by automatically inferring key program invariants and specifications: both dynamically, for non-linear invariants and array invariants, and statically, for program interface invariants (Section 2.2). Our approach is the first to tackle non-linear and array invariants, and our experimental results suggest that we can find half of the invariants needed for formal verification and that our invariants are comparable in quality to what humans would write (Section 3.2). Finally, we developed techniques for scaling GenProg to resource-constrained embedded systems (Section 2.3). Our approach works well at the assembly level as well as on program binaries and requires only 15% of the RAM and 5% of the disk space required by our pre-award work. Taken together, we feel that these results make a strong case for the applicability of automated program repair to new domains.

References

[1] Symantec. Internet security threat report. In *http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepape%r_symantec_internet_security_threat_report_x_09_2006.en-us.pdf*, September 2006.

[2] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, NIST, May 2002.

[3] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (to appear)*, 2012.

[4] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering (to appear)*, 2012.

[5] Raymond P. L. Buse and Westley Weimer. Synthesizing API usage examples. In *International Conference on Software Engineering (to appear)*, 2012.

[6] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automatic program repair through the evolution of assembly code. In *Automated Software Engineering*, pages 33–36, 2010.

[7] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., 1996.

[8] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] C. V. Ramamoothy and W-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.

[10] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, 2005.

[11] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming language design and implementation*, pages 141–154, 2003.

[12] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[13] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Automated Software Engineering*, pages 34–43, 2007.

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

[15] Michael Barr. Faulty code will lead to an era of firmware-related litigation. In *Electronic Design*, January 2010.

[16] CNN. Iraqi insurgents hacked Predator drone feeds, U.S. official indicates. In *http://www.cnn.com/2009/US/12/17/drone.video.hacked/index.html*, December 2009.

[17] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

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

[19] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Genetic and Evolutionary Computation*, pages 1427–1434, 2011.

[20] Brad L. Miller and David E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evol. Comput.*, 4(2):113–131, 1996.

[21] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference*, 2010.

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

[23] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26, 2005.

[24] Stephanie Forrest, Westley Weimer, ThanhVu Nguyen, and Claire Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference*, pages 947–954, 2009.

[25] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *International Conference on Genetic Algorithms*, pages 2–9, 1989.

[26] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall, 1993.

[27] Michael Karr. Affine relationships among variables of a program. *Acta Informitica*, 6:133–151, 1976.

[28] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[29] Ben Wegbreit. The synthesis of loop predicates. *Communication ACM*, 17(2):102–113, 1974.

[30] Steven M. German and Ben Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions Software Engineering*, 1(1):68–75, 1975.

[31] Shmuel Katz and Zohar Manna. Logical analysis of programs. *Communication ACM*, 19(4):188–206, 1976.

[32] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Principles of Programming Languages*, pages 132–143, 1977.

[33] Nachum Dershowitz and Zohar Manna. Inference rules for program annotation. In *International Conference on Software Engineering*, pages 158–167, 1978.

[34] M.D. Ernst. *Dynamically detecting likely program invariants*. PhD thesis, University of Washington, 2000.

[35] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *International Conference on Software Maintenance*, pages 736–743, 2001.

[36] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Symposium on Operating systems Principles*, pages 87–102, 2009.

[37] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astrée analyzer. In *European Symposium on Programming*, pages 21–30, 2005.

[38] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Languages Design and Implementation*, pages 196–207, 2003.

[39] Mardavij Roozbehani, Eric Feron, and Alexandre Megrestki. Modeling, optimization and computation for software verification. In *Hybrid Systems: Computation and Control*, pages 606–622, 2005.

[40] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model Checking and Abstract Interpretation*, pages 25–41, 2005.

[41] C. A. R. Hoare. Proof of a program: Find. *Communication ACM*, 14:39–45, January 1971.

[42] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Programming Language Design and Implementation*, pages 321–333, 2000.

[43] Edward Cohen. *Programming in the 1990s: an introduction to the calculation of programs*. Springer-Verlag, 1990.

[44] Peter Hallam. What do programmers really do anyway? In *Microsoft Developer Network — C# Compiler*, Jan 2006.

[45] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, NJ, USA, 2001.

[46] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.

[47] David G. Novick and Karen Ward. What users say they want in documentation. In *Conference on Design of Communication*, pages 84–91, 2006.

[48] R. Holmes, R. Cottrell, R.J. Walker, and J. Denzinger. The end-to-end use of source code examples: An exploratory study. In *International Conference on Software Maintenance*, pages 555–558, 2009.

[49] Rajeev Jain. API-writing and API-documentation. In `http://api-writing.blogspot.com/`, April 2008.

[50] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building more usable apis. *IEEE Softw.*, 15(3):78–86, 1998.

[51] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *International Conference on Computer documentation*, pages 133–141, 2002.

[52] Forrest Shull, Filippo Lanubile, and Victor R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Trans. Softw. Eng.*, 26(11):1101–1118, 2000.

[53] Jeffrey Stylos, Brad A. Myers, and Zizhuang Yang. Jadeite: improving api documentation using usage information. In *Extended Abstracts on Human Factors in Computing Systems*, pages 4429–4434, 2009.

[54] Martin P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Softw.*, 26(6):27–34, 2009.

[55] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, pages 318–343. 2009.

[56] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Towards an intelligent code search engine. In *AAAI Conference on Artificial Intelligence*, 2010.

[57] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Principles of Programming Languages*, pages 4–16, 2002.

[58] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Programming Languages Design and Implementation*, pages 48–61, 2005.

[59] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Automated Software Engineering*, pages 204–213, 2007.

[60] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *International Conference on Software Engineering*, pages 117–125, 2005.

[61] javadoctool@sun.com. How to write doc comments for the Javadoc tool. In *http://www.oracle.com/technetwork/java/javase/documentation/index-13786%8.html*, 2010.

[62] Oracle. Java SE 6 documentation. In *http://download.oracle.com/javase/6/docs/*, 2010.

[63] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, 2011.

[64] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, 2009.

[65] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

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

[67] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *Automated Software Engineering*, pages 313–316, 2010.

[68] Jason Fitzpatrick. An interview with Steve Furber. *Commun. ACM*, 54:34–39, May 2011.

[69] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[70] Linda G. Shapiro, George C. Stockman, Linda G. Shapiro, and George Stockman. *Computer Vision*. Prentice Hall, January 2001.

[71] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

[72] Enric Rodríguez Carbonell. *Automatic Generation of Polynomial Invariants for System Verification*. PhD thesis, Technical University of Catalonia (UPC), Barcelona, 2006.

[73] Xiang Yin, John C. Knight, and Westley Weimer. Exploiting refactoring in formal verification. In *Dependable Systems and Networks*, pages 53–62, 2009.

[74] Xiang Yin, John C. Knight, Elisabeth A. Nguyen, and Westley Weimer. Formal verification by reverse synthesis. In *Computer Safety, Reliability, and Security*, pages 305–319, 2008.

[75] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity*, volume 2021, pages 500–517, 2001.

[76] John Levon. *OProfile Manual*. Victoria University of Manchester, 2004.

6. List of Symbols, Abbreviations and Acronyms

- AES. Advanced Encryption Standard. A common mission-critical cryptographic algorithm.
- API. Application program interface. A set of functions and variables by which different software modules communicate. The correct use of an API is governed by rules (i.e., a specification).
- ASM. Assembly language program. A list of human-readable, architecture-specific machine instructions. Such a program can be assembled and linked to create a binary executable.
- AST. Abstract syntax tree. An internal representation for the human-readable source code (e.g., C, Java, ADA) of a program.
- ELF. Executable and linking format. A common container format for binary executable programs. No source code or assembly code is retained at this stage.
- GenProg. An automated program repair approach that is the subject of this award. It is based on genetic programming.
- LOC. Lines of Code. A common metric for software size or complexity.
- NLA. Non-Linear Arithmetic. A suite of programs that have non-trivial invariants, suitable for evaluating inference approaches.
- SMT. Satisfiability Modulo Theories. A general constraint system that is similar to, but more elaborate than, boolean satisfiability. Many off-the-shelf SMT solvers exist.