Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices

Eric Schulte* Jonathan DiLorenzo† Westley Weimer† Stephanie Forrest*

* Department of Computer Science
University of New Mexico
Albuquerque, NM 87131-0001
{eschulte,forrest}@cs.unm.edu

† Department of Computer Science
University of Virginia
Charlottesville, VA 22904-4740
{jd9hz,weimer}@cs.virginia.edu

Abstract

We present a method for automatically repairing arbitrary software defects in embedded systems, which have limited memory, disk and CPU capacities, but exist in great numbers. We extend evolutionary computation (EC) algorithms that search for valid repairs at the source code level to assembly and ELF format binaries, compensating for limited system resources with several algorithmic innovations. Our method does not require access to the source code or build toolchain of the software under repair, does not require program instrumentation, specialized execution environments, or virtual machines, or prior knowledge of the bug type.

We repair defects in ARM and x86 assembly as well as ELF binaries, observing decreases of 86% in memory and 95% in disk requirements, with 62% decrease in repair time, compared to similar source-level techniques. These advances allow repairs previously possible only with C source code to be applied to any ARM or x86 assembly or ELF executable. Efficiency gains are achieved by introducing stochastic fault localization, with much lower overhead than comparable deterministic methods, and low-level program representations.

When distributed over multiple devices, our algorithm finds repairs faster than predicted by naïve parallelism. Four devices using our approach are five times more efficient than a single device because of our collaboration model. The algorithm is implemented on Nokia N900 smartphones, with inter-phone communication fitting in 900 bytes sent in 7 SMS text messages per device per repair on average.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.5 [*Software Engineering*]: Testing and Debugging; D.3.2 [*Language Classifications*]: Macro and Assembly Languages; I.2.8 [*Artificial Intelligence*]: Heuristic methods

General Terms Experimentation, Languages

Keywords Automated program repair, evolutionary computation, fault localization, assembly code, bytecode, legacy software

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03. . . $15.00

1. Introduction

Automated software repair is an emerging research area in which algorithmic and heuristic approaches are used to search for, generate, and evaluate candidate repairs for software defects. It has received attention in programming languages (e.g., [15]), operating systems (e.g., [26]) and software engineering (e.g., [33, 35]) venues. Automated repair methods have been applied to multiple classes of software engineering and security defects (e.g., [35]) including hard-to-fix concurrency bugs [15], have won human-competitive awards [9], and automatic repairs have been successfully pitted against DARPA Red Teams to demonstrate quality [26]. With bug repair dominating software development costs (90% of the total cost of a typical software project is incurred after delivery [28]) such automated techniques are of increasing importance.

However, few automated repair techniques apply to resource constrained embedded systems, instead targeting desktop client software such as Firefox [15, 26], server software such as MySQL [15] or webservers, or design-by-contract Eiffel programs [33]. Given the tight coupling between embedded software and the unique execution environments in which they operate, desktop testing and repair tools are often insufficient. Current research in this field is increasingly out of step with the needs of industry, in which embedded microprocessors account for more than 98% of all produced microprocessors [4]. 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 [2]. In addition, previous repair techniques that apply to binaries [26] or assembly language [27] have uniformly targeted Intel x86, despite "the widespread dominant use of the ARM processor in mobile and embedded systems" [8].

Evolutionary computation (EC) is a stochastic search method based on Darwinian evolution, which has been applied to the automated repair problem. Like other search methods, EC is relevant when it is easier to evaluate a candidate solution than to predict the form of a correct solution [5].[1] Although EC techniques do not yet synthesize large programs in traditional languages, they can repair a wide variety of real defects at the source-code level in real-world software applications [20].

We propose to repair assembly language and executable binary programs directly using EC across multiple architectures, including embedded and mobile systems. Resource constraints in embedded

[1] In this paper we use the term genetic algorithm (GA) interchangably with evolutionary computation (EC). GAs and Genetic Programming (GP) are two concrete realizations of the general EC framework. GAs are typically defined over linear strings and GP typically refers to tree-based executable representations of programs. Our method uses linear strings that are executable.

and mobile systems preclude the use of existing techniques. For example, it is not feasible to trace all operands that CPU instructions read or write (as in [26, Sec. 2.1.1]) or even to trace all instructions visited (as in [35]). A pre-set, unified locking discipline may not be available (as used by [15]). Source code with debugging information may not be available, so statements and abstract syntax trees cannot be accurately identified to reduce search-space size (as used by [35]), and, formal pre- and post-condition annotations are unlikely to be present (as used by [33]). Finally, most embedded devices do not ship with a complete compiler toolchain, and a deployed device may not have the storage or RAM to support its original mission together with a heavyweight repair framework that uses the GCC toolchain. The closest assembly-level repair work is a preliminary four-page short paper [27] that proposes automated program repair on x86 assembly language, but does not address fault localization, multiple architectures, repairing executables, or distribution across multiple systems.

The main contributions of this paper are as follows:

- An architecture-independent representation and stochastic fault localization algorithm that supports automatic program repair at the assembly and binary levels. We demonstrate applicability to x86 and ARM processors. Although sampling for fault localization is not new, its application and evaluation in the domain of program repair is, and our technique is an order-of-magnitude faster and of comparable accuracy to deterministic methods.
- An empirical demonstration that disk space and memory requirements are 95.22% and 85.71% smaller, respectively, than similar methods, allowing application to mobile and embedded systems.
- An empirical comparison across different levels of program representation with respect to EC's success and efficiency at finding repairs.
- A demonstration that most source-statement-level repairs reported previously [35] can also be carried out at the assembly and binary level with a 62.43% average decrease in the time required to perform a repair between the AST and ELF levels.
- A distributed GA that performs automated program repair across multiple embedded devices. Four devices using our approach are five times faster than a single device for our repair benchmarks.
- Empirical demonstration of the repair method on Nokia N900 smartphones (600 MHz ARM Cortex-A8 CPU, 256 MB memory). Using two phones, the distributed algorithm conducts repairs with just under 900 bytes sent (or 7 SMS messages) per participant per repair.

2. Background

In automated software repair (e.g., [26, 33, 35]), defects are corrected by searching over and evaluating a solution space of possible repairs. The set of repairs (fixes) may be generated from random variations of the instructions in the program, as in GenProg [20]; from formal source code annotations, as in AutoFix-E [33]; or from a pre-defined library, such as the locking changes used by the AFix project [15] or the clamp-variable-to-value operation of Clearview [26].

Our work is based on the EC approach [9, 35, 36]. EC mimics Darwinian evolution in a computational algorithm that searches for candidate solutions to a given problem [14]. In the program repair context, a population of program variants is generated by applying random mutations to the original buggy source code. These variants are then compiled using the program's build toolchain, and the resulting executables are evaluated against the program's test suite. The test suite assesses the goodness, or *fitness*, of each variant, and the fitness value is used to *select* which variants will remain in the population, undergoing additional mutations. In addition to mutation, the algorithm uses *crossover* to exchange instructions between two variants, producing a recombination of partial solutions. The process iterates until a variant passes all test cases and also fixes the bug, or until an upper time limit is reached.

To reduce the size of the search space of possible repairs, mutation and crossover operators are limited to re-organizing statements present in the original program. No new statements are created, and sub-statement program elements, such as expressions, are not changed directly. Fault localization (e.g., [16]) focuses the mutation operators on statement nodes executed on the buggy input. To collect this information, an execution trace is recorded from an instrumented version of the program run against the test suite.

We extend this previous work to run directly on compiled assembly files and linked ELF executables, and we introduce a stochastic method of fault localization appropriate for these lower-level representations. These extensions remove the requirement for source code availability and the need for compilation and linking (for the ELF level) as part of the search process. It is applicable to arbitrary assembly and ELF programs rather than only C-language programs and enables repairs that are not expressible at the C statement level.

3. Technical Approach

Build processes generate intermediate representations, each of which is a possible target for automated repair, and each of which poses unique challenges for the repair method. These challenges include: the form of the representation and mutation operators, the granularity of fault localization information, and the tools required to express a representation as an executable program. The following subsection reviews the repair framework at a high level, which mirrors the AST algorithm [35, 36]. We summarize the technical and algorithmic aspects of ASM and ELF repairs, including the resources required by each level of representation, the effects of representation on mutation operations, and the requirements for expression as executable programs.

3.1 Evolutionary Repair Algorithm

The algorithm shown in Figure 1 applies to source-level ASTs, compiled ASM, and compiled and linked ELF repairs. Section 4.1 reports values for the parameters, such as popsize, used in our experiments. The next subsections present the stochastic fault localization algorithm, ASM and ELF representations, and mutation and crossover operators.

The overall structure in Figure 1 is iterative. Tournament selection selects variants for the next iteration (generation) (Lines 8, 12); retained variants exchange sub-strings to produce offspring (Line 9); all variants are mutated (Line 16), and the process repeats until a solution is found (Line 18).

3.2 Stochastic Fault Localization

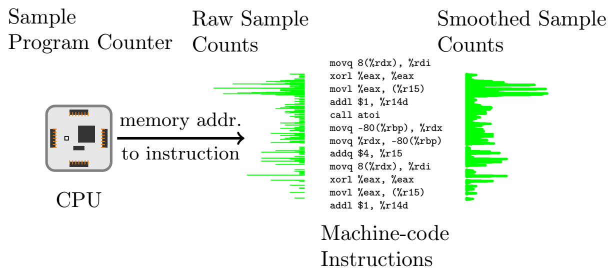
In large programs, it is reasonable to assume that most parts of the program are *not* related to a given bug [16], and fault localization methods often target code executed on the bug-inducing input. Accurate fault localization is critical for targeting the repair operators [35, Fig. 5] and is an important factor in running time [36, Fig. 1].

Our earlier work recorded entire sequences, or paths [35], of executed statements using various weighting factors [16], which required expensive program instrumentation or runtime harnesses. A key challenge is obtaining accurate enough information for guiding automated repairs, while maintaining the efficiency required for embedded devices.

```
Input: Program $P$ to be repaired.
Input: Set of positive testcases $p \in PosT$.
Input: Set of negative testcases $n \in NegT$.
Output: Repaired program variant $V$.
 1: $Path_{PosT} \leftarrow \bigcup_{p \in PosT}$ locations visited by $P(p)$
 2: $Path_{NegT} \leftarrow \bigcup_{n \in NegT}$ locations visited by $P(n)$
 3: $Path \leftarrow$ set_weights($Path_{NegT}, Path_{PosT}$)
 4: $Pop \leftarrow$ initial_population($P$, pop_size)
 5: repeat
 6:     $NewPop \leftarrow \emptyset$
 7:     for $i = 1 \rightarrow$ (pop_size $\times$ $cross\_percent$) by 2 do
 8:         $V_1, V_2 \leftarrow$ tournament($Pop$), tournament($Pop$)
 9:         $NewPop \leftarrow NewPop \cup$ {crossover($V_1, V_2$)}
10:     end for
11:     for $i = 1 \rightarrow$ (pop_size $\times$ $(1 - cross\_percent)$) do
12:         $NewPop \leftarrow NewPop \cup$ {tournament($Pop$)}
13:     end for
14:     $Pop \leftarrow \emptyset$
15:     for all $\langle V, Path_V \rangle \in NewPop$ do
16:         $Pop \leftarrow Pop \cup$ {fitness(mutate($V, Path_V$))}
17:     end for
18: until $\exists \langle V, Path_V, f_V \rangle \in Pop \mid f_V =$ max_fitness
19: return $V$
```

Figure 1. High-level pseudocode for EC-based automatic program repair, which applies to all levels of representation. Representation-specific subroutines such as fitness(V) and mutate(V, $Path_V$) are described subsequently.

Many traditional code profilers (e.g., `gcov`) are language specific and rely on the insertion of assembly instrumentation consuming unacceptable storage and run-time resources. For example, instrumentation of the `flex` benchmark [35] at the C-language level required storing sequential ordering information from about 443,399 raw statement visits, with program instrumentation increasing the CPU run-time by a factor of 100. Direct extensions [2] such as deterministic sampling of the program counter (e.g., using `ptrace`) performed poorly in our preliminary work.



```
movq 8(%rdx), %rdi
xorl %eax, %eax
movl %eax, (%r15)
addl $1, %r14d
call atoi
movq -80(%rbp), %rdx
movq %rdx, -80(%rbp)
addq $4, %r15
movq 8(%rdx), %rdi
xorl %eax, %eax
movl %eax, (%r15)
addl $1, %r14d
```

Figure 2. Stochastic Fault Localization (raw and smoothed samples from the `merge-cpp` benchmark).

To address these constraints, we propose a sampling approach to fault localization (Figure 2), which is applicable to arbitrary assembly and ELF programs and dramatically reduces resource requirements compared to earlier work. We sample the program counter (PC) across multiple executions of the program. These sampled memory addresses are then mapped to bytes in the `.text` section of ELF files or to specific instructions in ASM files. The result is a count of the total number of times each instruction in the program was sampled. Stochastic sampling only approximates control flow and is vulnerable to gaps, elided periodic behavior, over-sampled instructions, etc. To overcome these limitations, we apply a 1-D Gaussian convolution to the sampled addresses with a radius of 3 assembly instructions, s.t. the smoothed value of each sample $G(x)$ is a weighted sum of the raw value $F(x)$ of itself and its 6 nearest neighbors.

$$G(x) = \sum_{i=-3}^{3} F(x+i) \times \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}i^2}$$

This simple transformation increases the chance that instructions that are executed but not directly sampled will be counted, and it improves the correlation between stochastic and deterministic samples (Section 4.2).

Gaussian convolution is an accepted method of smoothing data to reduce detail and noise in fields such as computer vision [29]. However, to our knowledge it has not previously been applied to fault localization. Section 4.2 compares the fault localization information produced by our stochastic sampling to a fully deterministic program trace.

Samples are collected from multiple runs of each of the program's tests. Despite multiple executions of each test, the CPU time required is less than comparable deterministic techniques (Section 4.4). The union of every instruction included in the fault localization from the positive tests, and the union of every instruction included in the fault localization from the negative tests are collected into the positive and negative fault localization respectively. These sets of are then used to guide program repair.

3.3 ASM and ELF Program Representations

In contrast to the tree-structured (nested) source code representation [35], our assembly and ELF level representations use a linear sequence of instructions e.g., as produced by `gcc -S` or `objdump -d -j .text` respectively. Candidate repairs are generated by swapping, inserting or deleting instructions. Pseudo-operations and directives at the assembly level (e.g., `.section .rodata`), and all elements outside of the `.text` section of the executable at the ELF level, are retained in each variant but never modified by mutation. To reduce search space size, each instruction, together with its operands, is treated atomically, and operands are not mutated independently. Both representations are source language and architecture agnostic.

3.3.1 Genetic Operators

ASM and ELF representations are modified using four operators designed for the linear representations: three mutation types and one crossover type (Figure 3). Instructions appearing in the negative localization are ten times more likely to be mutated than instructions appearing in both.

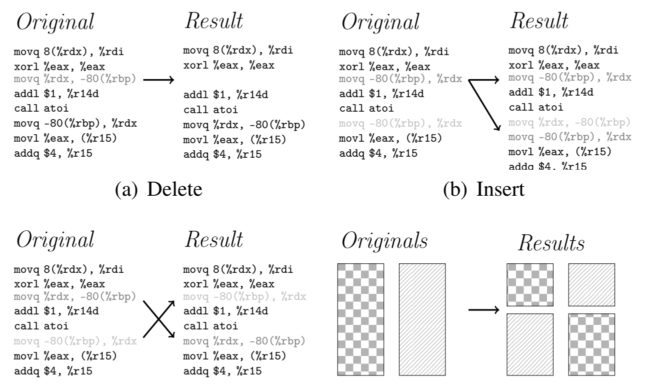
Linked ELF executables often contain hard-coded memory addresses included as literals in the program `.text`. Since there is no general way to distinguish an integer literal from an address literal, our mutation operators occasionally create invalid addresses (recent work may enable improved treatment of literal addresses [1] in the future).

To minimize disturbance of literal addresses and information outside of the `.text` section, the ELF mutation and crossover operators attempt to maintain the length of the linear instruction array in bytes and minimize changes to the in memory addresses of existing instructions.

Delete: A single instruction selected by weight is removed. At the ELF level, every byte of the deleted instruction is replaced with a `nop` byte.

Insert: A single instruction selected at random is copied to a new location selected by weight. At the ELF level, changes to offsets

[2] Code available at http://github.com/eschulte/tracer.



```
movq 8(%rdx), %rdi
xorl %eax, %eax
movq %rdx, -80(%rbp)
addl $1, %r14d
call atoi
movq -80(%rbp), %rdx
movl %eax, (%r15)
addq $4, %r15
```

```
movq 8(%rdx), %rdi
xorl %eax, %eax

addl $1, %r14d
call atoi
movq %rdx, -80(%rbp)
movl %eax, (%r15)
addq $4, %r15
```

(a) Delete

```
movq 8(%rdx), %rdi
xorl %eax, %eax
movq -80(%rbp), %rdx
addl $1, %r14d
call atoi
movq -80(%rbp), %rdx
movl %eax, (%r15)
addq $4, %r15
```

```
movq 8(%rdx), %rdi
xorl %eax, %eax
movq -80(%rbp), %rdx
addl $1, %r14d
call atoi
movq %rdx, -80(%rbp)
movq -80(%rbp), %rdx
movl %eax, (%r15)
addq $4, %r15
```

(b) Insert

```
movq 8(%rdx), %rdi
xorl %eax, %eax
movq %rdx, -80(%rbp)
addl $1, %r14d
call atoi
movq -80(%rbp), %rdx
movl %eax, (%r15)
addq $4, %r15
```

```
movq 8(%rdx), %rdi
xorl %eax, %eax
movq -80(%rbp), %rdx
addl $1, %r14d
call atoi
movq %rdx, -80(%rbp)
movl %eax, (%r15)
addq $4, %r15
```

(c) Swap

(d) Crossover

Figure 3. Genetic operators for ASM. ELF mutations have the same form but manipulate bytes instead of text instructions. They also use padding to minimize length changes.

are minimized by removing a number of nearby `nop` instruction equal to the size of the inserted code. In the rare case where there are insufficient `nop` instructions, the size of the `.text` section increases, which usually reduces individual fitness.

Swap: An instruction selected at random is swapped with an instruction selected by weight. This operation is naturally length-preserving and requires no special treatment at the ELF level, although swapping two instructions of different lengths may alter instruction offsets in the short region between those two locations.

Crossover: Given two parent programs (Figure 1, line 11), a single index less than the length of the shorter parent program, is selected at random. Single-point crossover is performed, concatenating the instructions from one parent up to the selected index, and the instructions from the other parent after the selected index. This operation produces two new variant programs. At the ELF level, the index is selected such that the number of bytes before the index is the same in each parent.

3.3.2 Fitness Evaluation

Before fitness evaluation, the variant must be converted to an on-disk executable. Generating an executable from an ASM representation requires writing the array of ASM instructions (and any assembly directives or pseudo-operations) to disk and assembled/linked using instructions taken from the program's original build sequence. The ELF representation is written directly to a binary executable ELF file on disk without any elements of the software project's build toolchain.

The executable is then run against the program's test suite and negative test case. To protect against dangerous behavior by the random variants, a lightweight sandboxing solution was readily constructed from standard Linux utilities (e.g., `ulimit` and `chroot`). All experimental results presented in the paper include the cost of sandboxing. Repairs to kernel-level embedded code that manipulate hardware directly would require different methods. Fitness is assessed by running the executable on all test cases, computing a weighted average score based on the passing test cases, where negative test cases count twice as much as positive ones.

3.4 Motivating Example

Consider the program for exponentiation shown in Figure 4, which illustrates the expressive power of ASM- and ELF-level repairs. It contains a bug in which the two arguments are assigned to the wrong variables.

Buggy Exponentiation Program

```c
int main(int argc, char *argv[]) {
  double a,b,c;
  a = atoi(argv[2]); // should be argv[1]
  b = atoi(argv[1]); // should be argv[2]
  c = 1;

  while(b > 0) {
    c = c * a;
    b--;
  }

  printf("%g\n", c);
  return 0;
}
```

Figure 4. Exponentiation with transposed arguments. This program cannot be repaired by simple reorganization of extant C statements.

The repair for this program should assign `atoi(argv[1])` to `a` and `atoi(argv[2])` to `b`. This is not possible using only combinations of extant statements (as in [35]), because each assignment is atomic and subexpressions may not be altered. However, when operating at the lower levels of compiled assembly instructions or linked executables, the repair may be expressed trivially through the transposition of two `mov` instructions. In fact, this program was repaired at the ASM and ELF levels with 3842 and 6453 test executions respectively, but it was not repaired in over 1000 runs of the algorithm at the AST level. In addition to bugs such as this, which are not repairable using combinations of existing atomic source-level statements, other bugs are markedly easier to repair at the ASM and ELF levels (Section 4.3).

3.5 Distributed Genetic Algorithm (DGA)

The methods described in Section 3.2 and Section 3.3 enable automated repair in devices with severely limited disk and memory resources. For example, the Nokia smartphones used in our experiments provide only 256 MB of memory. Also, such devices may not be fast enough to find and evaluate a successful repair in a reasonable amount of time on their own. To address these concerns, we present a distributed genetic algorithm (DGA) that allows multiple devices to collaborate in finding a repair.

As illustration, we consider a group of mutually trusting smartphones cooperating to repair the same bug. The repair may be found more quickly if the search burden can be distributed over many devices. This is a plausible use case given the large number of homogeneous installs of smart phone applications.[3]

The DGA is considered successful, compared to naïve parallelism in which all devices work independently, if the total number of fitness evaluations required to find a repair is reduced, thus reducing time and power costs. A second design goal is to minimize network communication. In the smartphone scenario, we assume that communication occurs via infrequent SMS messages, rather than high-power, high-bandwidth links.

Distributed GAs [17] are based on the insight that separate genetic populations sharing a small amount of data can often outperform a single population of the same total size. Each participating device maintains its own population of variants and periodically shares high-fitness variants with other devices. GAs are known to

[3] In theory, repairs from an untrusted source could be self-certifying (cf. [3]). In this paper we consider only networks of trusted, uncompromised devices.

be more effective when they operate over genetically diverse populations [18], and there is a large literature on the problem of "premature convergence" in GAs (e.g., [10]). Because the search in each sub-population is dominated by local high-fitness variants, diverse sub-populations on each device can explore different parts of the search space in parallel. Performance is thus enhanced by maximizing diversity among the sub-populations stored on each device. We hypothesize this to contribute to our DGA's superlinear reduction in fitness evaluations over naïve parallelism (see Section 4.5).

Two novel aspects of the DGA for software repair are: splitting the fault-localization search space among devices, and diversity-based migration.

3.6 Splitting the Search among Participants

To maximize sub-population diversity, we use fault localization information to constrain the search such that each device explores a different region of the search space. Recall that the repair algorithm modifies only those parts of the program identified by fault localization (Section 3.2) and that positive fault localization instructions are weighted differently from negative instructions with respect to choosing mutation locations. Let S be the ordered list of atomic elements of the program representation (e.g., assembly instructions, or groups of bytes) identified by fault localization over the positive test cases. Given N devices, we assign each device responsibility for two contiguous sub-sequences of S of size $k = \frac{|S|}{N}$, although each device contains the entire program and a list of the statements visited by negative test cases only (Section 3.2). We hypothesize that contiguous statements are likely relevant to the same repair. Formally, if $s_j \in S$ is the j^{th} element of S counting in representation order, then device i of N only modifies elements of S_i where:

$$S_i = \left\{ s_j \in S \;\middle|\; \begin{array}{l} i \bmod N \leq \frac{s_j}{k} < (i+2) \bmod N \\ \vee\ visited\_on\_negative\_tests\_only(s_j) \end{array} \right\}$$

Note that since the insert and swap operators take one operand from the fault localization weighting and one at random, this division does not formally partition the search space, but it does divide the work of searching it into slightly overlapping parcels.

3.7 Diversity-based Migration

Each device periodically communicates a subset of its current population to a subset of the other devices. We hypothesize that search performance is improved if diverse variants are shared. We measure diversity between variants as the number of unique edits in their representations. An iterative calculation identifies the n most diverse variants in a subpopulation. Each variant is assigned one point for each genetic change that is unique across the (initially empty) output set; the candidate with the most points is placed in the output set, with ties broken randomly (this is common in the first generation, when the output set is empty); and the process repeats until the output set contains the desired number of variants.

3.8 Distributed Algorithm Details

Given these methods for dividing the search among different devices, and for identifying diverse variants to share among devices we now formalize the DGA algorithm in Figure 5. The DGA is executed concurrently by N networked nodes, each of which starts with the same information—program to repair, test suite, network topology (for exchanging information), and fault localization. Each node then creates a local initial subpopulation of variants (lines 2–4; Figure 1) and carries out one generation of the repair algorithm.

Each node selects a diverse set of d variants from its subpopulation (Figure 5, line 6). The exact migration topology does not generally affect algorithmic performance [6]. To minimize communication, we use a simple permutation topology, which changes each generation: Each participant passes variants to its "right" neighbor

```
Algorithm: Distributed Repair
Input: Program $P$ to repair.
Input: Set of positive testcases $p \in PosT$.
Input: Set of negative testcases $n \in NegT$.
Input: Number $d$ of variants per migration.
Input: Number $N$ of networked participants.
1: $Subpop \leftarrow \mathsf{initial\_population}(P, \mathsf{pop\_size})$
2: for $generation = 1 \rightarrow \mathsf{gen}$ do
3:     $Id \leftarrow$ temporary device specific network identifier
4:     $Subpop \leftarrow \mathsf{run}(Subpop, PosT, NegT)$     (Fig. 1, 6–17)
5:     $Migrants \leftarrow \mathsf{div\_select}(Subpop, d)$     (Sec. 3.7)
6:     $\mathsf{send}(succ(Id), Migrants)$
7:     $Migrants \leftarrow \mathsf{receive}(pred(Id))$
8:     $Subpop \leftarrow Subpop \cup Migrants$
9: end for
```

Figure 5. Distributed genetic algorithm (DGA) for program repair. The search is distributed among participants that share information (diverse high-fitness program variants) after each generation.

(line 6) in the permutation, receiving a similar set from its "left" neighbor (line 7). The incoming variants are added to each node's subpopulation and are subject to selection in the next generation. The process then repeats. When participant finds a repair, the repair is sent in an out-of-band message, and the process terminates early (not shown).

The number of variants exchanged is at most $N \times d \times \mathsf{gen}$. Since d is chosen to be small, and the number of generations is small, communication cost is effectively linear in the number of participants. Because all variants share a common ancestor in the original program, only the edit history needs to be communicated (cf. [20, Sec. III-B]). For example, a variant created by deleting instruction 3 and then swapping instructions 1 and 2, can be serialized in a form such as "d(3)s(1,2)". In practice, we encode the operation (delete, insert, or swap) in one byte and operation-specific operands in one or two 16-bit integers. Fitness is included as a final byte. This encoding assumes self-contained compact descriptions of edits, and thus does not admit crossover.

4. Experimental Results

This section presents empirical results evaluating the ASM and ELF representations and the DGA. The results show the following:

1. Stochastic fault localization closely approximates the deterministic approach (Section 4.2).
2. Repair success at the ASM and ELF representation levels is similar to that reported previously for ASTs (Section 4.3).
3. The ASM and ELF representations, together with stochastic fault localization, have small resource footprints, suitable for running on embedded devices (Section 4.4).
4. The DGA increases success rates while reducing total fitness evaluations required to find a repair (Section 4.5).

4.1 Experimental Setup

Table 1 lists the benchmark defective programs evaluated in this paper. For ease of comparison they are taken from earlier work [35] with two additions. `merge` sort was added to evaluate the stochastic fault localization algorithm on a test suite with full assembly statement coverage, and `merge-cpp` was added to demonstrate a language other than C. Each program comes with a regression test suite, used to validate candidate repairs, and at least one test case indicating a defect. These programs have on average $3.69\times$ more assembly instructions and $9.55\times$ more bytes in the `.text` section of ELF files than lines of source code.

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
merge-cpp	71	421	1540	merge sort (in C++)	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	31862	117682	304288		

Table 1. Benchmark programs used in experiments, taken from Weimer *et al.* [35] with the addition of `merge`. Program size is reported as follows: 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. Each program has a regression test suite and a failing test case indicating a fault.

We used the following GA parameters: population size popsize $= 1000$; maximum number of fitness evaluations in a trial evals $= 5000$, mutation rate mut $= 1.0$ per individual per generation and crossover rate cross $= 0.5$ crossovers per indivdiual per generation.

Most experiments were run on a machine with 2.2 GHz AMD Opteron processors and 120 GB of memory. The wall-clock evaluation of the DGA was conducted on a single server-class machine, with 3.0 GHz Intel Xeon CPUs and 15.6 GB of memory. Cell phone experiments were conducted on Nokia N900 smartphones, each of which features a 600 MHz ARM Cortex-A8 CPU and 256 MB of mobile DDR memory.

4.2 Fault Localization Evaluation

We collected program counter samples using `oprofile` [21], a system-wide profiler for Linux systems, which doesn't alter the profiled program, runs on embedded devices, and when appropriately configured, has minimal impact on system-wide performance.

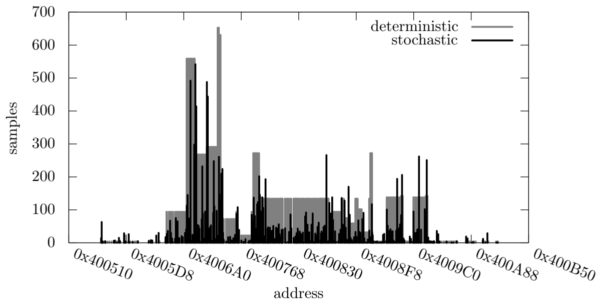
We process the samples as described in Section 3.2, explicitly comparing to deterministic approaches (below) and evaluating utility in the context of program repair in Section 4.3. The direct comparison used `merge` sort, which is small and exhaustively tested with 100% statement and branch coverage, as well as `deroff`, which is larger and has less-complete test coverage. The stochastic and deterministic traces taken from the failing test cases of both programs are shown in Figure 6. Inputs for the failing test case exercise the bugs described in Table 1.

Ten stochastic samples and one deterministic sample were collected for both programs. We find high correlations of 0.96 (merge) and 0.65 (deroff) among the stochastic samples, indicating consistency across samples. We find lower correlations of 0.61 (merge) and 0.38 (deroff) between the naïve stochastic (no Gaussian convolution) and deterministic samples, which increase to 0.71 (merge) and 0.48 (deroff) after Gaussian convolution, indicating that smoothing provides significant improvement.

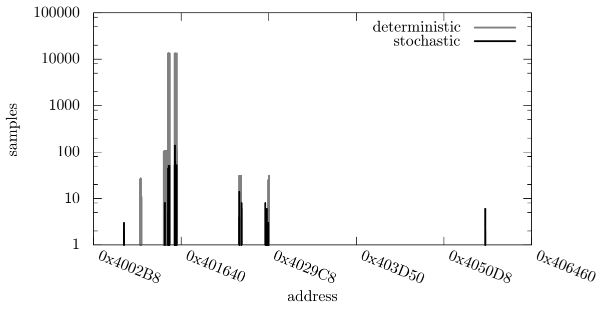
4.3 ASM and ELF Repair Success

Table 2 compares ASM and ELF to AST, averaging over 100 trials of the standard (non-DGA) algorithm for each tested configuration. Memory usage, which varies insignificantly across runs, was calculated from a single run.

Our Nokia smartphones have 256+768 MB of RAM plus swap. In that environment, only 8 of 13 programs can be repaired at the



(a) `merge`



(b) `deroff`

Figure 6. Fault localization in program address space. Results for stochastic sampling shown in *black* identify program regions similar to those found deterministically, shown in *gray*.

AST level (i.e., memory is exhausted or the repair fails) compared to 10 at the ASM level and 11 at the ELF level.

We expected the repair process to be much less efficient, especially at the ELF level, both in terms of success rate and time to first repair. Generally, EC searches are more challenging in larger search spaces—more locations to choose for a mutation and more possible instructions to choose from when performing an insertion. However, 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. Using Fisher's Exact test to compare success rates we find no significant

Program	Memory (MB) AST	Memory (MB) ASM	Memory (MB) ELF	Runtime (s) AST	Runtime (s) ASM	Runtime (s) ELF	% Success AST	% Success ASM	% Success ELF	Expected Fitness Evaluations AST	Expected Fitness Evaluations ASM	Expected Fitness Evaluations 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
`merge-cpp`	†	50	60	†	121.87	90.56		100	79	†	314.00	2135.2658
`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
w/o `units`	1242	559	135	229.50	278.20	74.02	77.00	76.00	67.18	583.98	188.38	207.15

Table 2. Resource comparison of abstract syntax tree (AST) assembly source (ASM) and ELF binary (ELF) repair representations. "Memory" reports the average max memory required for a repair (as reported by the Unix `top` utility). "Runtime" reports the average time per successful repair in seconds. "% Success" gives the percentage of random seeds for which a valid repair is found within 5000 runs of the full test suite. "Expected Fitness Evaluations" counts the expected number of evaluations per repair (Equation 1). † Indicates that there were no successful repairs in 5000 fitness evaluations. Rows with † are excluded when calculating "Memory", "Runtime" and "Expected Fitness Evaluations" averages. * Indicates memory requirements in excess of the 1024 available on the Nokia smartphones.

difference between AST and either ASM or ELF with p-values of 1 (between AST and ASM), and 0.294 (between AST and ELF). This suggests that automated repair at the ASM and ELF levels is a practical alternative to source-level repair.

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

The `atris` repair involves deleting a call to `getenv`. At the AST level this requires a single deletion, while at the ASM level, three contiguous instructions must be deleted and the repair was not found. The repair was found at the ELF level, and all of the five repairs found were unique, each involving from 3 to 7 accumulated mutation operations.

The `merge` repair involves replacing an `if` statement with its `else` branch. At the AST level this requires swapping exactly those two statements, which is 1 of 4900 possible swap mutations. At the ASM and ELF levels, modification of a single comparison instruction suffices to repair the program. This is one of only 218 such changes possible and is much more easily found by our technique.

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

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

Given that repair time is dominated by fitness evaluation, which includes compilation and linking 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 usually more efficient at the ASM and ELF levels than at the AST level.

4.4 Resource Requirements

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. Section 4.5 also evaluates network communication for the DGA.

CPU Usage. Runtime costs associated with GA 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 and fitness evaluation (including compilation and linking depending upon the representation used).

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, AST-level repairs require compilation of an instrumented program with a $100\times$ slowdown per run (Section 3.2). Related executable-level approaches introduce a $300\times$ slowdown to compute fault localization information [26, 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.

After fault localization is complete, both ASM and ELF representations have lower fitness evaluation costs than the AST-level because compilation and linking are not required. However, the search problem for ASM and ELF is potentially much larger than for AST (compare the size columns in Table 1), suggesting they may need more fitness evaluations to find a repair. If the time to conduct a repair at the AST level is normalized to 1.0, on average ASM repairs take $7.22\times$ and ELF repairs take $0.38\times$. Using a Mann-Whitney U-test the runtime difference between ELF and AST is a significant improvement, with a p-value of 0.055. While the ASM-level repair is slower, this can be mitigated through collaboration across multiple devices (Section 4.5).

Memory. Memory utilization is important for mobile and embedded devices. The earlier work was conducted on server machines with 8 GB [26] to 16 GB of RAM [35]. By contrast, the Nokia N900 smartphones we consider as indicative use cases have 256 MB Mobile DDR—an order of magnitude less.

Table 2 reports the memory used (in MB) for repairs at the AST, ASM and ELF representations, showing 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 stores only the `.text` section of ELF binaries in memory.

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

AST requires the source code and build tool chain of the original program. Our baseline comparison, GenProg, 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 implementation is currently incorporated into the AST repair framework [35] 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. Although these are merely implementation details, we argue that such conditions and the need to minimize the on-disk footprint are indicative of many embedded devices.

4.5 Distributed and Embedded Repair Results

Table 3 summarizes the performance of the DGA with the number of nodes ranging from one to four. The "% Success" column lists the fraction of trials for which a successful repair is found, normalized so that a single non-networked participant has 1.0. Overall success rate improves by 13% from one to four participants because they share diverse variants and collaborate on the search by exploring different portions of the program space.

In most instances, the time to find the first repair is critical. The "Expected Fitness Evals" column measures that effort in a machine-independent manner (Section 4.3). In practice, fitness evaluations (which require repeatedly running the program test suite) account for the majority of algorithmic runtime. The number of fitness evaluations required to find a repair drops by a factor of 5 and the average standard deviation by 62%. Each fitness evaluation includes the time to run the test suite of the subject program.

	% Success				Expected Fitness Evals			
	$\rightarrow$ #nodes $\rightarrow$				$\rightarrow$ #nodes $\rightarrow$			
Program	1	2	3	4	1	2	3	4
`atris`	1	1.00	1.00	1.00	1	0.53	0.40	0.27
`ccrypt`	1	1.00	1.00	1.00	1	0.62	0.27	0.24
`deroff`	1	1.00	1.00	1.00	1	0.58	0.44	0.32
`flex`	1	1.13	1.87	2.07	1	0.87	0.46	0.40
`indent`	1	1.04	1.04	1.04	1	0.25	0.16	0.10
`look-s`	1	1.00	1.00	1.00	1	0.50	0.55	0.29
`look-u`	1	1.00	1.00	1.00	1	0.76	0.37	0.32
`merge`	1	1.69	2.14	2.31	1	0.43	0.22	0.18
`s3`	1	1.00	1.00	1.00	1	0.47	0.31	0.24
`uniq`	1	1.00	1.00	1.00	1	0.49	0.46	0.31
`units`	1	1.27	1.33	1.33	1	0.32	0.11	0.07
`zune`	1	1.00	1.00	1.00	1	0.47	0.36	0.27
h.mean	1	1.07	1.12	1.13	1	0.47	0.28	0.20

Table 3. Distributed repair results as a function of the number of participant nodes. "Success Rate" gives the percentage of random trials that produce a repair, normalized to 1.0 for 1 node. "Expected Fitness Evals" approximates the time and effort required by the first machine to find a repair and is defined in Section 4.3; it is normalized to 1.0 at 1 node. As the number of participants grows from 1 to 4, repairs are found 13% more often and the number of fitness evaluations required decreases by a factor of 5 (harmonic mean of all benchmarks). The results listed in this table are from runs at the AST level and thus `merge-cpp` is not included.

As a baseline, we also measured the performance of naïve parallelism (i.e., using two Nokia cell phones to run two separate copies of the repair algorithm and stopping when the first repair is found) to demonstrate that the distributed algorithm is responsible for the performance gains. For a meaningful comparison, we focused on benchmarks that take more than one generation to repair (i.e., `flex`, `indent`, `merge` and `units`) and thus reached the distributed portion of the DGA algorithm described in Figure 5. Over 100 trials, two nodes running DGA use only 442 total fitness evaluations per repair compared to 848 for naïve parallelism (not counting evaluations performed after the first repair is found). On two nodes, DGA thus requires 48% fewer fitness evaluations than naïve parallelism.

Finally, we calculate the network bandwidth consumed by DGA. Recall that at the end of each generation, each machine sends $d = \mathsf{popsize}/20 = 50$ diverse variants to a neighbor. After the first generation, each variant has one edit in its history ($\mathsf{mutp} = 1.0$). One third of all mutations are deletions, which can be represented by four bytes (the opcode and the operand and the fitness). Insertions and swaps require six bytes (the opcode, two operands, and the fitness). The expected data size sent from one participant to another after the first generation is thus $50 \times (4 \times \frac{1}{3} + 6 \times \frac{2}{3}) = 267$ bytes. After the second generation, each of the d variants sent will typically have two edit operations in its history, follow the same distribution (and thus require twice as much bandwidth to communicate). Since each of the N networked participants send one batch of variants after every generation, the expected cumulative total network bandwidth used, as a function of the number of generations G before the repair is found, is estimated as:

$$\mathsf{bytes\_sent}(G) \simeq \sum_{i=1}^{G} N \times d \times i \times (4 \times \frac{1}{3} + 6 \times \frac{2}{3})$$

With our default parameters, this estimate implies 801 bytes sent per participant (spread over two network sends or six SMS messages per participant) by the end of the second generation. Empirically, we find that our measured results match this model to within 10%. For example, over eight random trials to repair `merge` with two participants averaging 2.4 generations, DGA sent a total of 7000 bytes in such a way that 51 SMS text messages were required. This implies that just under 875 bytes (or 6.4 SMS text messages) were required per node per trial. We claim that this low communication cost is well within what would be considered reasonable for the task of program repair across embedded or mobile systems.

To measure the impact of communication on run time we next evaluated wall-clock times for the DGA, the original serial algorithm, and an ideal naïve parallel adaptation of the original algorithm with no inter-node communication, all at the AST source code level. These runs were performed on a single server-class machine, with 4 3.0 GHz Intel Xeon CPUs and 15.6 GB of memory using TCP over Ethernet. The DGA incurs a cost from inter-node communication, but the results in Table 4 show that this cost is more than offset by the increased algorithmic efficiency of search space splitting and migration of high-fitness variants among the nodes.

Table 4 shows the mean wall clock time in seconds to find a repair for the DGA and a naïve parallel version of the original serial algorithm. We report results for one to four participating repair nodes (in practice all nodes were run in parallel on a single multicore). The naïve algorithm running on a single node is exactly the original serial algorithm.

Table 4. Mean wall clock time in seconds to find a successful repair at the AST source code level. Communication in the DGA is bulk synchronous. The column "Rounds" gives the number of communication rounds. Results are averaged over 10 attempted repairs of the `units` program. The other three programs that require multiple generations to repair show the same trend with from 1 to 4 nodes, and using DGA or naïve parallel repair algorithm.

	DGA		Naïve Parallel
# Nodes	Seconds	Rounds	Seconds
1			205.531
2	173.868	43.2	195.821
3	135.17	28.2	201.346
4	115.566	14.5	211.989

At each level of parallelization, the distributed algorithm finds repairs faster than the ideal naïve parallelization of the original algorithm. Judging the significance of these differences using the Kolomogorov-Smirnov test (used instead of a T-test because these distributions are not normal given the large differences between runs which do or do not find a repair) yields a value of 0.5 with $p = 0.000214$. The wall clock performance gained by using the distributed algorithm over a naïve parallel algorithm is statistically significant.

5. Related Work

Genetic Algorithms and Evolution of Machine Code. Previous work can be divided into two broad categories. One designs mutation operators to preserve the validity of the programs they manipulate, both over CISC instruction sets [23] running on hardware and more recently over Java byte code [13, 25] running on the Java virtual machine. An alternative is to use generic mutation operators, relying on safety mechanisms built into the CPU and operating system to catch and terminate invalid individuals [19].

Our work follows the second approach, using general operators that can be applied across both CISC x86 and RISC Arm architectures. Given the low cost of linking ASM individuals and writing ELF individuals to disk, we favor using the CPU to check validity, finding that it is adequate.

"Evolvability" analysis [24] has led some to declare x86 assembly code an unfit medium for evolutionary computation [31]. Based on this belief, translations between x86 and more evolvable intermediate languages have been proposed, both to construct evolvable malware [7] and to distribute binary patches [1]. Our results provide a counterexample, showing that EC at the ASM and ELF level can efficiently generate viable program variants. By operating on whole instructions, the negative effects of "argumented instructions" [24] are minimized. Similarly, through the use of `nop` padding in ELF level mutation, changes in the absolute offset are minimized, thus reducing the impact of direct addressing.

Assembly Program Repair. Schulte *et al.* provide the closest instance of related work proposing automated program repair for assembly programs [27]. This preliminary short paper describes EC-generated program repairs to x86 assembly programs, focusing only on x86 assembly as a lowest common denominator for high-level languages (such as C and Haskell). Here, we extended that work to target both ASM and ELF representations, multiple assembly languages (x86 and ARM), propose stochastic fault localization, demonstrate that our technique scales to embedded devices, and propose and demonstrate a novel distributed repair algorithm.

Executable Program Repair. The closest instance of related work that automatically repairs binary executables is the Clearview system [26], which patches errors in deployed Windows x86 binaries. Clearview uses a split-phase learning-and-monitoring approach—program instrumentation is used to learn invariants, and later monitoring notices violations in deployed programs. If a violation occurs, Clearview considers possible patches, evaluating them against an indicative workload of test cases. Clearview focuses on specific error types and prespecifies a set of repair templates (e.g., breaking out of a loop, clamping a variable to a value, etc.). It is therefore less general than our method, although it has obtained impressive results in its domain, patching nine out of twelve historical Firefox vulnerabilities, even in the face of a DARPA Red Team. However, the Clearview approach seems heavy weight; they report experiments involving rack-mount server machines with 16 GB of RAM, VMware virtualization, and a 300× slowdown during the learning phase [26, Sec. 4.4]. By contrast, our work targets resource-constrained embedded environments and uses stochastic sampling to reduce the cost of fault localization, which does not require program instrumentation.

A distributed repair technique has also been described for Clearview which protects application communities, but that work did not report performance results [26]. Their distributed algorithm amortizes the cost of learning invariants (i.e., of computing fault localization information), and ignores the time to find a candidate repair. By contrast we demonstrate a distributed algorithm that finds repairs more quickly.

Distributed Evolutionary Algorithms. The large literature on distributed and parallel genetic algorithms dates back to Grosso [11], who explored the idea of subdividing a GA population into smaller subpopulations with occasional exchanges of fit individuals among the populations. More recent work ranges from implementations tailored to particular hardware configurations [12, 22] to a wide variety of algorithms in which the population is partitioned and individuals are shared among the partitions according to different schemes, e.g., [22]. Parameter tuning (population size, migration rate, etc.) is a concern, with recommendations available for several problem types [6]. Perhaps most relevant to the current work is a Doctoral Colloquium describing a distributed genetic programming implementation on a wireless sensor network [32], although that work is still quite preliminary.

6. Discussion

Compared to automated program repair over C statements, an assembly representation operates over a finite alphabet of elements. A typical assembly instruction consists of an opcode and two or three operands, while C statements may be of arbitrary size and complexity (e.g., `x = 1 + 1 + ...`). In addition, there are typically at least three times more assembly instructions than C statements. These combined facts give an assembly representation limited to permutations of elements of the original program much higher sample rates in the space of possible programs. In a system where the linking of an assembly file does not introduce new instructions or arguments, the alphabet and expressive power of the ASM and ELF representations are equivalent.

We see the effects of this increased coverage in Section 3.4 in which the program is only repaired at the ASM and ELF levels, and in Section 4.3 in which some repairs are much more easily expressed at the ASM and ELF levels.

While introducing new bugs is a possibility, We typically find very small changes which address only the buggy behavior. Specifically we reviewed the repairs presented here and found no evidence of introduced bugs.

While moving from the tree AST representation to the vector ASM and ELF representations may seem minor, the EC community has two separate sub-fields, GP and GA, dedicated to the study of tree- and vector-based representations respectively, each with their own research challenges (e.g., bloat in GP), application domains, best practices, journals and conferences.

There were several technical challenges in the implementation, particularly regarding the manipulation of ELF files. Existing tools such as the GNU ELF tool suite (`libfd`) and its BSD equivalent (`libelf`) do not support changes to the contents of existing ELF files. We thus developed our own libraries for manipulating ELF files, including support for the automated updates to ELF file metadata in response to an altered `.text` section[4].

Although ELF files do support symbolic addressing through symbol names and run-time linkers, direct addresses pose a significant problem for the randomly changing raw binary code sections. This is mitigated by mutation operators that minimize disruption to the location of compiled code.

It is sometimes useful to consider the total number of unique repairs produced. For example, additional candidate repairs help developers create high-quality final patches [34]. Because our DGA explicitly manages diversity across sub-populations, we hypothesize that it might produce a wider variety of distinct repairs. We measured uniqueness in terms of changes made to the code: Two repairs are distinct if they use edit operations, treated as unordered sets, that are not identical. With four participants, our DGA found 20% more unique repairs than with one participant. If we consider only the challenging `flex`, `indent`, `merge` and `units` repairs, the the number of discovered unique repairs increases to 73%.

Some may argue that it is aesthetically unappealing to modify code at all, particularly with a stochastic algorithm such as EC. We believe that distributed automated repair methods will be necessary in the future, as embedded devices become ubiquitous and are deployed in a wide range of environments. As the computational power of distributed embedded devices eclipses that of centralized servers, timely centralized testing and repair becomes infeasible (cf. already 28–29 day lag times are reported in recent surveys for centralized repairs [30]).

There are several areas of potential future work. For example, we expect that variations of this technique could be used to optimize performance of programs for specific environments. A second area is proactive diversity to disrupt software monocultures. In security settings, randomization is often inserted into compiled programs to prevent malicious attacks. Assembly code evolution could be used to add diversity to deployed software.

[4] Code available at: http://github.com/eschulte/rw-elf and http://github.com/eschulte/elf.

6.1 Limitations and Caveats

The fine granularity of repairs at the ASM and ELF levels may be a poor match for conventional test suites. For example, we have observed ASM-level repairs that change the calling convention of one particular function. Such a repair has no direct representation at the C source level, and a test suite designed to maximize statement coverage (for example) may not speak to the validity of such a repair. Producing efficient test suites that give confidence that an implementation adheres to its specification remains an open problem in software engineering. Our work shares this general weakness with all other approaches that use test suites or workloads to validate candidate repairs (e.g., Clearview [26] and GenProg [35]). In this regard, sandboxing is crucial: we have observed ASM variants that subvert the testing framework by deleting key test files, leading to perfect fitness for all subsequent variants until the test framework is repaired.

Benchmark selection is a threat to the external validity of our experiments. We used benchmarks taken from published papers to admit direct comparison; to mitigate this threat we augmented the benchmark set with very high test coverage and non C-language examples.

Our DGA uses a self-contained compact encoding of each variant, to facilitate communication among nodes via SMS. This encoding precluded the use of crossover because it would not be meaningful to exchange information between two variants under the encoding, even though crossover is an important feature of many GAs. This restriction implies that our results might not generalize to other GAs. Since crossover can improve search time and success rates, it is possible that our results would be improved with a DGA implementation that supports crossover. This could be tested using a recently published *patch* representation [20] that supports a concise encoding of crossover.

7. Summary and Conclusion

This paper extends previous work on automated program repair at the AST level to compiled (ASM) and linked (ELF) programs. The new representations allow repairs when source code cannot be parsed into ASTs (e.g., due to unavailable source files, complex build procedures or non-C source languages). They also reduce memory and disk requirements sufficiently to enable repairs on resource constrained devices. We also introduce a stochastic fault localization technique, which is applicable to these representations and devices, and present a distributed repair algorithm that allows costly repair processes to be split across multiple devices.

Importantly for embedded devices, our techniques reduce memory requirements by up to 85%, disk space requirements by up to 95% (Section 4.4), and repair generation time up to 62% (Section 4.4), which enables application to resource-constrained environments. We demonstrate our technique on Nokia N900 smartphones whose resource constraints which serve as a practical proxy for future low power computing systems.

Our fault localization algorithm is based on stochastic sampling and Gaussian convolution. It provides the instruction- and byte-level precision required by the ASM and ELF representations, while retaining sufficient accuracy to guide automated repair. In addition, it is ten times faster than previous approaches and more suited to devices where direct instrumentation is infeasible.

We take advantage of these reduced resource requirements in a distributed repair algorithm in which multiple cell phones communicate via SMS messages to find repairs more quickly. Sections 3.8 and 4.5 detail the algorithm. Using four devices we increase success rates by 13% and reduce fitness evaluation burdens by a factor of five—a superlinear improvement over naïve parallelism. The distributed algorithm's use of multiple populations could also be used to speed up serial repair on a single device. Communication costs are low: two phones require under 900 bytes (or 7 SMS messages) per participant per repair on our benchmarks.

Taken together, these techniques constitute the first general automated method of program repair applicable to binary executables, and are a first step in the application of automated software repair to the growing field of mobile and embedded devices.

7.1 Acknowledgments

C. Le Goues, N. Holtschulte, R. Bodik, and the anonymous reviewers provided helpful comments on the manuscript. The work was supported by DARPA (P-1070-113237), DOE (DE-AC02-05CH11231), NSF (SHF-0905236, SHF-0905373, CCF-0954024) and the Santa Fe Institute.

References

[1] S. Adams. Google software updates: Courgette (design documents). `http://dev.chromium.org/developers/design-documents/software-updates-courgette`, 2011.

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

[3] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worm epidemics. *ACM Trans. Comput. Syst.*, 26(4):1–68, 2008. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/1455258.1455259.

[4] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.

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

[6] F. Fernández, M. Tomassini, and L. Vanneschi. An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):21–51, 2003.

[7] P. Ferrie. Malware analysis: Flibi night. *Virus Bulletin*, pages 4–5, March 2011.

[8] J. Fitzpatrick. An interview with Steve Furber. *Commun. ACM*, 54: 34–39, May 2011. doi: http://doi.acm.org/10.1145/1941487.1941501. URL `http://doi.acm.org/10.1145/1941487.1941501`.

[9] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computing Conference*, 2009.

[10] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16 (1):122–128, 1986.

[11] P. B. Grosso. *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model*. PhD thesis, The University of Michigan, Ann Arbor, MI, 1985.

[12] S. Harding. Genetic programming on graphics processing units bibliography. Memorial Univeristy, Canada, Feb. 9, 2010. `http://www.gpgpgpu.com`.

[13] B. Harvey, J. A. Foster, and D. A. Frincke. Towards byte code genetic programming. In *Genetic and evolutionary computing conference*, page 1234, 1999.

[14] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, 1992. Second edition.

[15] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, pages 389–400, 2011.

[16] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005. ISBN 1-59593-993-4. doi: http://doi.acm.org/10.1145/1101908.1101949.

[17] Z. Konfrst. Parallel genetic algorithms: Advances, computing trends, applications and perspectives. In *Parallel and Distributed Processing Symposium*. IEEE, 2004.

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

[19] F. Kühling, K. Wolff, and P. Nordin. Brute-force approach to automatic induction of machine code on CISC architectures. In *European Conference on Genetic Programming*, pages 288–297, 2002. ISBN 3-540-43378-3.

[20] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, 2012.

[21] J. Levon. *OProfile Manual*. Victoria University of Manchester, 2004.

[22] A. Munawar, M. Wahib, M. Munetomo, and K. Akama. A survey: Genetic algorithms and the fast evolving world of parallel computing. In *High Performance Computing and Communications*, pages 897–902, 2008. ISBN 978-0-7695-3352-0. doi: http://doi.ieeecomputersociety.org/10.1109/HPCC.2008.77.

[23] P. Nordin, W. Banzhaf, and F. D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In *Advances in Genetic Programming 3*, pages 275–299. June 1999. ISBN 0-262-19423-6. URL `http://www.aimlearning.com/aigp31.pdf`.

[24] C. Ofria, C. Adami, and T. C. Collier. Design of evolvable computer languages. *IEEE Transactions on Evolutionary Computation*, 6:420–424, 2002.

[25] M. Orlov and M. Sipper. Genetic programming in the wild: evolving unrestricted bytecode. In *Genetic and evolutionary computation*, pages 1043–1050, 2009. ISBN 978-1-60558-325-9.

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

[27] E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *Automated Software Engineering*, pages 313–316, 2010.

[28] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003. ISBN 0321118847.

[29] L. G. Shapiro, G. C. Stockman, L. G. Shapiro, and G. Stockman. *Computer Vision*. Prentice Hall, 2001. ISBN 0130307963.

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

[31] S. Thomas. Taking the redpill: Artificial evolution in native x86 systems. `http://spth.vxheavens.com/ArtEvol.html`, October 2010.

[32] P. Valencia. In situ genetic programming for wireless sensor networks. In *SenSys Doctoral Colloquium*, 2007.

[33] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[34] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, 2006.

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

[36] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53 (5):109–116, 2010.