Automated Repair of Binary and Assembly Programs across Groups of Embedded Devices

withheld

ABSTRACT

Mobile and embedded systems have limited memory, disk and CPU capacities but exist in great numbers. We present a method for automatically repairing arbitrary software defects in embedded systems. 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 any prior knowledge of the bug to be repaired.

We repair defects in ARM and x86 assembly as well as ELF binaries, and observe a decrease of 86% in memory and 95% in disk requirements, with a 62% decrease in repair time, compared to similar source-level approaches. These efficiency gains are due to novel techniques: approximate stochastic fault localization, which has much lower overhead than comparable deterministic methods, and low-level program representations. These advances allow repairs previously only applicable to C source code to be applied to any ARM or x86 assembly or ELF executables.

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 [**Software Engineering**]: Macro and Assembly Languages

General Terms

Experimentation, Languages

Keywords

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

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., [20]), operating systems (e.g., [32]) and software engineering (e.g., [39, 42]) venues. Automated repair methods have been applied to multiple classes of software engineering and security defects (e.g., [8, 42]) including hard-to-fix concurrency bugs [20], have won human-competitive awards [13], and automatic repairs have been successfully pitted against DARPA Red Teams to demonstrate quality [32]. With bug repair dominating software development costs (90% of the total cost of a typical software project is incurred after delivery [34]) such automated techniques are of increasing importance.

However, few automated repair techniques apply to mobile or embedded systems, instead targeting desktop client software such as Firefox [20, 32], server software such as MySQL [20] or webservers [8], or design-by-contract Eiffel programs [39]. Given the tight coupling between embedded software and the unique execution environments in which these systems operate desktop testing and repair tools are often insufficient. The weight of current research is increasingly out of step with the needs of industry, in which embedded microprocessors account for more than 98% of all produced microprocessors [6]. Failures in such devices have both civilian and military implications, ranging from faulty-firmware lawsuits [2] to insurgents hacking Predator drone feeds [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 [3]. In addition, previous repair techniques that apply to binaries [32] or assembly language [33] have uniformly targeted Intel x86, despite "the widespread dominant use of the ARM processor in mobile and embedded systems" [12].

Evolutionary computation (EC) is a general stochastic search strategy 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 [7].[1] Although evolutionary techniques are as yet unable to create new programs from whole cloth, recent work has shown them capable of repairing a wide variety of real defects in real-world software applications [25].

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 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 [32, Sec. 2.1.1]) or even to trace all instructions visited (as in [42]). A pre-set, unified locking discipline may not be available (as used by [20]). 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 [42]) and formal pre- and post-condition annotations are unlikely to be present (as used by [39]). 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 [33] that proposes automated program repair on x86 assembly language, but does not cover fault localization, multiple architectures, repairing executables, or scaling to mobile devices.

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.
- 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 ability and efficiency at finding repairs.
- A demonstration that most source-statement-level repairs reported previously [42] 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.
- 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 on an example repair task.
- Empirical demonstration of our approach on Nokia N900 smartphones (600 MHz ARM Cortex-A8 CPU, 256 MB memory). Using two phones, our distributed algorithm conducts repairs with just under 900 bytes sent (or 7 SMS messages) per participant per repair.

[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 typically refer to evolutionary algorithms defined over linear strings (the approach taken in this paper), and GP typically refers to evolutionary algorithms that work on tree-based representations of programs.

2. BACKGROUND

In automated software repair (e.g., [32, 39, 42]), 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 [25]; from formal source code annotations, as in AutoFix-E [39]; or from a pre-defined library, such as the locking changes used by the AFix project [20] or the clamp-variable-to-value operation of Clearview [32].

Our work is most closely related to previous work that demonstrated how to repair multiple classes of errors at the source level using EC [13, 41, 42]. These approaches mimic Darwinian evolution in a computational algorithm that searches for candidate solutions to a given problem [18]. In the program repair context, a population of program variants is generated by applying random mutations to the 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.

To reduce the size of the search space of possible repairs, previous work limits mutation and crossover operators to the re-organization of statements present in the original program. No new statements are introduced by the algorithm, and sub-statement program elements, such as expressions, cannot be changed directly. Fault localization techniques (e.g., [21]) focus the mutation operators on statement nodes that are 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. This extension removes the requirement for source code availability and the need for compilation and linking(when repairing ELF executables) as part of the search process. It is applicable to arbitrary assembly and ELF programs rather than only to C-language programs and makes possible repairs which can not be expressed at the C statement level.

3. TECHNICAL APPROACH

Build processes have several associated intermediate representations, each of which is a possible target for automated repair and each of which poses unique challenges for the repair method. These include: the form of the representation and the associated 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 evolutionary repair algorithm at a high level, which mirrors that of the AST algorithm first described by Weimer *et al.* [41, 42]. 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 all program representations considered in this paper (i.e., source level AST, compiled ASM, and compiled and linked ELF). Section 4.1 reports values for the parameters, such as popsize, used in our experiments. The next subsections describe a new fault localization algorithm the ASM and ELF representations, and mutation and crossover operators.

```
Input:  Program $P$ to be repaired.
Input:  Set of positive testcases $PosT$.
Input:  Set of negative testcases $NegT$.
Output:  Repaired program variant.
 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: $Path \leftarrow \mathsf{set\_weights}(Path_{NegT}, Path_{PosT})$
 4: $Pop \leftarrow \mathsf{initial\_population}(P, \mathsf{pop\_size})$
 5: repeat
 6:     $NewPop \leftarrow \emptyset$
 7:     for $i = 1 \rightarrow (\mathsf{pop\_size} \times cross\_percent)$ do
 8:         $NewPop \leftarrow NewPop \cup \{\mathsf{tournament}(Pop)\}$
 9:     end for
10:     for $i = 1 \rightarrow (\mathsf{pop\_size} \times cross\_percent^{-1})$ by 2 do
11:         $V_1, V_2 \leftarrow \mathsf{tournament}(Pop), \mathsf{tournament}(Pop)$
12:         $NewPop \leftarrow NewPop \cup \{\mathsf{crossover}(V_1, V_2)\}$
13:     end for
14:     $Pop \leftarrow \emptyset$
15:     for all $\langle V, Path_V \rangle \in NewPop$ do
16:         $Pop \leftarrow Pop \cup \{\mathsf{fitness}(\mathsf{mutate}(V, Path_V))\}$
17:     end for
18: until $\exists \langle V, Path_V, f_V \rangle \in Pop \mid f_V = \mathsf{max\_fitness}$
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.

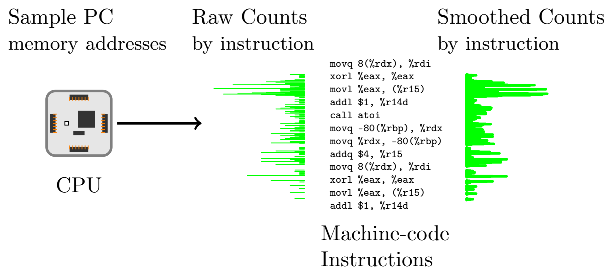
The overall structure in Figure 1 is iterative. Tournament selection is used to select variants for retention (Line 8); retained variants exchange sub-strings to produce offspring (Line 12); all variants are mutated (Line 16) and the process repeats until a solution is found (Line 18).

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 [21]. Accurate fault localization is thus critical for targeting the repair operators [42, Fig. 5] and is an important factor in running time [41, Fig. 1]. Common fault localization methods often 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 [42] of executed statements using various weighting factors [21] often requiring expensive program instrumentation or runtime harnesses. Our key challenge is to obtain information that is accurate enough to guide automated repairs but inexpensive enough to be gathered on embedded devices.

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 [42] at the C-language level stored sequential ordering information from about 443,399 raw statement visits, with program instrumentation increasing the CPU run-time by a factor of 100.4×. Direct extensions such as deterministic sampling of the program counter (e.g., using `ptrace` [2]) did not perform adequately 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 shown)

To address these constraints, we propose a sampling approach to fault localization (Figure 2) which is applicable to arbitrary assembly and ELF programs and has dramatically lowered resource requirements. 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 and 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 the results are often insufficient to guide the repair process because of 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 blurred 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 number of instructions implicated by each sample, decreases the impact of oversampling of specific instructions, and 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 [35], however, to the author's knowledge it has not previously been applied to fault localization. In Section 4.2 we compare the fault localization information produced by our stochastic sampling approach to a fully deterministic trace of the program execution.

3.3 ASM and ELF Program Representations

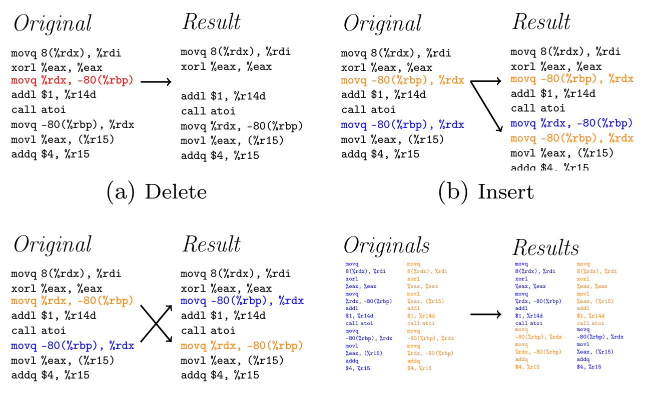
In contrast to the tree-structured source code statement level representation used in related work [42], our assembly and ELF level representations use a linear sequence of instructions (e.g., as produced by `gcc -S` or `objdump -d -j .text` respectively). Candidate repairs are generated

[2]Tool available under open source licensing at *withheld*

through swapping, copying, duplicating 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 but not modified. To reduce search space size, each instruction together with its operands is treated atomically, and operands are not mutated independently. Both of these representations are source language and architecture agnostic.

3.3.1 *Genetic Operators*

We modify the ASM and ELF representations using four operators designed for the linear representations: three types of mutation and one type of crossover. These operators are illustrated in Figure 3 and described below. Instructions are mutated based on a weighted product of the fault localization recorded from the positive and negative test cases.



(a) Delete — Original:

```
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
```

(a) Delete — Result:

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

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

(b) Insert — Original:

```
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
```

(b) Insert — Result:

```
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
```

(c) Swap — Original:

```
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
```

(c) Swap — Result:

```
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
```

(d) Crossover — Originals:

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

(d) Crossover — Results:

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

(a) Delete (b) Insert (c) Swap (d) Crossover

Figure 3: ASM representation genetic operators. ELF representation operators have the same form, but manipulate bytes instead of text instructions and use padding bytes to minimize length changes.

Linked ELF executables are 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, such references are left unchanged (however, recent work in this area may allow modification of such literal addresses [1] in the future).

To minimize the disturbance of literal addresses and of information outside of the `.text` section, the ELF mutation and crossover operations attempt to maintain the length of the linear instruction array in bytes and to 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 are minimized by removing a number of nearby `nop` instruction equal in size to the inserted code. In the rare case where there are insufficient `nop` instructions, the size of the `.text` section increases, which normally 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 betwen 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. This is not necessarily true of every index when instructions can have variable-length (such as in x86).

3.3.2 *Fitness Evaluation*

Before fitness evaluation, the representation 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. The resulting file is then assembled nad linked using the linking instructions taken from the program's original build sequence. The ELF representation can be written directly to a binary executable ELF file on disk and no elements of the software project's build toolchain are required.

In either case the resulting executable is then run against the program's test suite. To protect against pernicious evolved behavior a simple test harness limits the resources available to the evolved executable. Fitness is assessed by running the executable against the program's test suite and computing a weighted average of passing and failing test cases.

3.4 Motivating Example

This section uses a small example program to illustrate the increased expressive power of the ASM and ELF program representations. Consider the program for exponentiation shown in Figure 4. It contains a bug in which the two arguments are assigned to the wrong variables.

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 (e.g., as done by the GenProg technique [42]), 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 trivially be expressed through the transposition of two `mov` instructions. In addition to bugs such as this which are simply not repairable using combinations of existing atomic source-level statements, there are a number of bugs which are markedly easier to repair at the ASM and ELF levels. Such bugs are discussed in the experimental results (Section 4.3).

3.5 Distributed Genetic Algorithm (DGA)

The advances described in the Section 3.2 and Section 3.3 allow automated automated repair over mobile and embedded devices with severely limited disk and memory resources.

Buggy Exponentiation Program

```
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. Illustrates repair using the ASM and ELF program representations which manipulate machine-code instructions. This program is not repairable by related techniques which are limited to the recombination of extant C statements.

For example, the Nokia smartphones used in our experiments only provide 256 MB of memory. In addition to storage restrictions, a single embedded device may not be fast enough to find and evaluate a successful repair in a reasonable amount of time. To address this concern, we propose a distributed genetic algorithm (DGA) enabling multiple resource-constrained devices to collaborate in the discovery of a repair.

We consider scenarios such as a group of smartphones working together to fix a bug. In this case, a repair may be found more quickly if the search burden is distributed over many devices. Our DGA is intended for a setting in which several mutually trusting resource-constrained devices seek to repair the same bug—a plausible use case given the large number of homogeneous installs of smart phone applications.[3]

A distributed repair algorithm is successful, compared to naïve parallelism in which all devices work independently, if the number of fitness evaluations required to find a repair is reduced. This is beneficial in that it reduces the time (and thus 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 assuming high-power, high-bandwidth links.

Distributed GAs [22] 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 fit variants with other devices. GAs are known to be more effective when they operate over genetically diverse populations [23], and there is a large literature on the problem of "premature convergence" in GAs (e.g., [14]). Because the search in each sub-population is guided by local high-fitness variants, diverse sub-populations can potentially lead each device to investigate different parts of the search space in parallel. Performance is enhanced by maximizing diversity among the sub-populations stored on each device. We hypothesize this to be the reason our DGA achieves a super-linear speedup over naïve parallelism (see Section 4.5).

Two novel aspects of our DGA specific to the problem domain of automated bug repair are the division of the fault-localization defined search space across devices, and a method of migration tailored to a notion of diversity specific to our program transformations. The remainder of this section formalizes the details of our algorithm.

3.6 Splitting the Search among Participants

To maximize sub-population diversity, we use fault localization information to constrain the search in such a way that each device explores a different region of the search space. Recall that the repair algorithm modifies only those parts of the program indicated by fault localization (Section 3.2). Let S be the ordered list of atomic elements of the program representation (e.g., assembly instructions, or groups of bytes) weighted by fault localization over the positive test cases. Given N devices we assign each device two contiguous sub-sequences of S of size $k = \frac{|S|}{N}$ as well as all statements visited only by negative test cases. We hypothesize that contiguous statements are likely relevant to the same fix. 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:

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

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 does divide the work of searching it into slightly-overlapping parcels.

3.7 Diversity Selection

Each device periodically communicates a subset of its current population to a subset of the other devices: we hypothesize that search performance is improved when diverse variants are shared. We estimate diversity between variants by the number of unique changes that have been made to their representations. To find the most diverse candidates we perform the following iterative calculation. Each variant is assigned one point for each change in its genetic history which is unique across the (initially empty) output set. The candidate with the greatest number of points is selected and placed in the output set. If there are multiple candidates with the same score, one is chosen at random (this is common in the first generation, when the output set is empty). This process is repeated until the desired number of variants have been added to the output set.

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 distributed repair algorithm is executed concurrently by N networked nodes, each of which starts with the same information—program to repair, test suite, sequence of node permutations (i.e., network topology), and fault localization. Each node then creates a local initial population of

[3]In theory, repairs from an untrusted source could be self-certifying (cf. [5]). In this paper we consider only networks of trusted, uncompromised devices.

variants (lines 2–4; Figure 1) and carries out one generation of the evolutionary repair algorithm.

```
Algorithm:  Distributed Repair
Input:  Program $p$ to repair.
Input:  Fitness $f$ : candidate $\rightarrow \mathcal{N}$.
Input:  History $h$ : candidate $\rightarrow$ operation list.
Input:  Number $d$ of variants to distribute.
Input:  Number $N$ of networked participants.
 1: let $id$ = my unique network identifier
 2: let $P$ = initial population                          (Sec. 3.1)
 3: for $generation = 1 \ldots \mathsf{gen}$ do
 4:     let $\pi$ = permutation of $1 \ldots N$
 5:     $P, h \leftarrow$ one_generation($P, f, h$) (Sec. 3.1, steps 6–17)
 6:     let $tosend$ = diversity_select($P, f, h, d$)       (Sec. 3.7)
 7:     send $tosend$ to participant # (successor of $id$ in $\pi$)
 8:     let $added$ = receive from # (predecessor of $id$ in $\pi$)
 9:     $P \leftarrow P \cup added$
10: end for
```

Figure 5: Distributed genetic algorithm (DGA) for program repair over mobile and embedded devices. The search is distributed among participants that share information (program variants) after each generation. All nodes compute the same sequence of permutations π.

Each node selects a diverse set of d variants from this population (Figure 5, line 6). To minimize communication costs, we use a simple ring permutation topology: Each participant passes variants to its "right" neighbor (line 7), receiving a similar set from its "left" neighbor (line 8). The exact migration topology does not generally affect algorithmic performance [9]. The incoming variants are added to each node's population and are subject to selection in the next generation. The process then repeats. If any participant finds a repair, it sends that repair 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 most repairs are limited to few generations, the amount of communication is effectively linear in the number of participants. Because all variants share a common ancestor in the original program, only the edit history of a variant need be communicated between participants (cf. [25, 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 use a bare encoding in which the operation (delete, insert, or swap) is represented by one byte, the operation-specific operands are represented by one or two 16-bit integers, and the fitness is included as a final byte. This bare encoding requires self-contained compact descriptions of edits and thus does not admit crossover.

4. EXPERIMENTAL RESULTS

This section presents results of an empirical evaluation of 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. 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 4.4)
4. Our DGA increases success rates while reducing 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 the purposes of direct comparison the benchmarks are taken from related work on AST-level repair [42] with the addition of `merge` sort, which was added to evaluate the stochastic fault localization algorithm on a test suite with full assembly statement coverage and `merge-cpp`, which was added to demonstrate repair of a language other than C. Each program comes equipped 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 more assembly instructions and 9.55 more ELF bytes than lines of source code.

We used the following GA 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.

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 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` [26]. A system-wide profiler for Linux systems, which do not alter the program being sampled and, when appropriately configured, has a minimal impact on system-wide performance and may be used on embedded devices.

We process these samples as described in Section 3.2. We evaluate the resulting fault localization in the context of program repair in Section 4.3 and also explicitly compared the results of our fault localization and related deterministic approaches. For this comparison we 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 4.2.

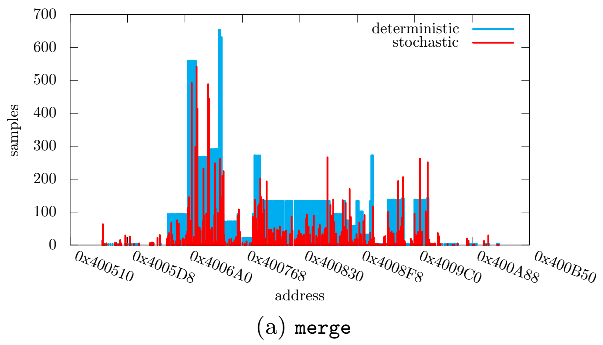
Ten stochastic samples and one deterministic sample were taken for each program. We find high correlations of 0.96 (merge) and 0.65 (deroff) between stochastic samples, indicating consistency across samples. We find lower correlations of 0.61 (merge) and 0.38 (deroff) between the naïve stochastic (no Gaussian convolution) and deterministic samples, which increase to 0.71 (merge) and 0.48 (deroff) after Gaussian convolution, indicating that our localization technique provides a significant improvement.

4.3 ASM and ELF Repair Success

Table 2 compares ASM and ELF to the AST representation used previously [42]. The reported results average over 100 trials for each tested configuration. Memory usage, which varies insignificantly across runs, was calculated from a single run.

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.* [42]. `merge` program was added because of its full-coverage test suite (Section 4.2). 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. Each program has a regression test suite and a test case indicating a fault.



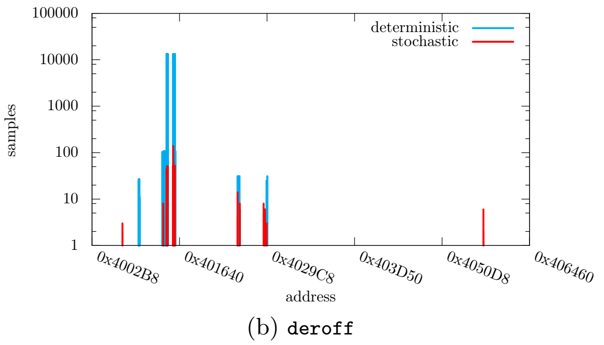


Figure 6: Fault localization in program address space. The stochastic results are shown in *red*; they identify similar program regions as related deterministic approaches, shown in *blue*.

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 that embedded environment, only 8 of 13 programs can be repaired at the AST level (i.e., exhaust memory or cannot be repaired at that level of representation) compared to 10 at the ASM level and 11 at the ELF level.

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. Using Fisher's Exact test to compare success rates we find little difference between AST and either ASM or ELF with p-values of 1 (between AST and ASM), and 0.294 (between AST and ELF).

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 and the repair was not found. The repair was found at the ELF level, 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 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 repre-

	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
`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

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.

sentation, we conclude that, when repairs are possible, the repair process is 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).

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, related AST-level work requires compilation of an instrumented program with a 100× slowdown per run (Section 3.2). Related executable-level approaches introduce a 300× slowdown to compute fault localization information [32, Sec 4.4], and `ptrace` full deterministic tracing incurs a 1200× slowdown. Our fault localization approach is an order-of-magnitude faster than these previous approaches.

For post-localization repair runtime, the ASM and ELF representations have lower fitness evaluation costs than AST-level approaches due to their lack of need for compilation and linking. However, for the same program, the search space for ASM and ELF is larger than the corresponding source-level search space (size columns in Table 1). If the time to conduct a repair at the AST level is 1.0, on average ASM repairs take 7.22× and ELF repairs take 0.38×. 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 performance hit may be mitigated through collaboration across multiple devices (Section 4.5).

Memory. Memory utilization is especially important for mobile and embedded devices. Previous approaches report results on repairs conducted on server machines with 8 GB [32] to 16 GB of ram [42]. 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. 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 makes it possible to repair embedded programs which are 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 GenProg AST-level evolutionary repair tool [42], 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 [42] 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 conditions and the need to minimize the on-disk footprint are indicative of many mobile and embedded devices.

4.5 Distributed and Embedded Repair Results

Table 3 summarizes the performance of our DGA as the number of nodes ranges 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 exploring different portions of the program space.

In many instances, the time taken to find the first repair is critical. The "Expected Fitness Evals" column measures that effort requirement in a machine-independent manner (Section 4.3). The number of fitness evaluations required to find a repair drops by a factor of 5 (harmonic mean; with the average standard deviation also dropping by 62%). Each fitness evaluation includes the time to run the test suite of the subject program.

As a baseline, we also measured the performance of naïve parallelism (i.e., using two machines to run two separate copies of our repair algorithm and stopping when the first finds a repair) to demonstrate that the distributed algorithm is responsible for the performance gains. For a meaningful comparison, we focused on benchmarks that required more than one generation (i.e., `flex`, `indent`, `merge` and `units`) and thus reached the distributed portion of the DGA algorithm described in Figure 5. Over 100 trials, we find that two nodes running DGA use only 442 fitness evaluations per repair compared to 848 using naïve parallelism. On two

	% Success				Expected Fitness Evals			
	→ #nodes →				→ #nodes →			
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 which 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 (taking the harmonic mean of all benchmarks).

nodes, DGA thus requires 48% fewer fitness evaluations than naïve parallelism.

Finally, we measure 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 information sent from one participant to another after the first generation is thus $50 \times (4 \times {}^{1}\!/_{3} + 6 \times {}^{2}\!/_{3}) = 267$ 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 given by:

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

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%. As a concrete example, over eight random trials to repair `merge` with two participants averaging 2.4 generations, our approach sends a total of 7000 bytes in such a way that 51 SMS text messages were required. That breaks down to just under 875 bytes (or

6.4 SMS text messages) per participant per trial. We claim that this such a low communication cost is well within what would be considered reasonable for the task of program repair across embedded or mobile systems.

5. RELATED WORK

Genetic Algorithms and Evolution. Previous approaches to the evolution of machine code can be broken into two broad categories. One category takes great care to craft mutation operators which preserve the validity of the programs they manipulate. This has been done over CISC instruction sets [29] running on hardware and more recently over Java byte code [17, 31] running on the Java virtual machine. The alternate approach is to use simple mutation operators and rely on the safety mechanisms built into the CPU and operating system to catch and terminate invalid individuals [24].

This work follows the second approach, using general genetic operators which 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 claim that the process of leveraging the CPU to check validity is not expensive enough to justify the use of more complex validity-preserving mutation operators.

"Evolvability" analysis [30] has lead some to declare x86 assembly code to be an unfit medium for evolutionary computation [37]. As a result, translations between x86 and more evolvable intermediate languages have been used to both construct evolvable malware [11] and also more efficiently distribute binary patches [1].

Our work provides a counterexample, showing that evolution at the ASM and ELF level can efficiently generate viable program variants. By operating on whole instructions, the negative effects of "argumented instructions" on evolution [30] 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 that applies automated program repair to assembly programs [33]. In their preliminary paper, they describe evolutionary program repairs to x86 assembly programs, focusing on x86 assembly as a lowest common denominator for high-level languages (such as C and Haskell). By contrast, we explicitly target both ASM and ELF representations, multiple assembly languages (x86 and ARM), propose a stochastic fault localization method, demonstrate that our technique scales to mobile devices, and demonstrate a distributed repair algorithm.

Executable Program Repair. The closest instance of related work that automatically repairs binary executables is the Clearview system [32], 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. Unlike this approach Clearview is not general and prespecifies a set of repair templates (e.g., breaking out of a loop, clamping a variable to a value, etc.). Clearview obtained impressive results, patching nine out of twelve historical Firefox vulnerabilities, even in the face of a DARPA Red Team. However, Clearview is somewhat 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 [32, Sec. 4.4]. By contrast, our approach specifically targets resource-constrained mobile 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 [32]. Their distributed algorithm amortizes the cost of learning invariants (i.e., of computing fault localization information), and not on reducing the time required 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 [15], 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 [16, 27] to a wide variety of distributed algorithms in which the population is partitioned and individuals are shared among the partitions according to different schemes, e.g., [27]. Parameter tuning (population size, migration rate, etc.) is a concern, with recommendations available for several problem types [10]. Perhaps most relevant to the current work is a Doctoral Colloquium describing a distributed genetic programming implementation on a wireless sensor network [38], although that work is still quite preliminary.

6. DISCUSSION

Compared to previous evolutionary program repair work that operates 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 an order-of-magnitude more assembly instructions than C statements. These combined facts give an assembly representation limited to permutations of elements of the original program a much higher coverage of 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 a program is only repairable 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.

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 meta-data in response to an altered `.text` section. These libraries are available under open source licensing.[4]

[4] *URL elided for blind submission*

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

An additional useful metric is the total number of unique repairs produced. Additional candidate repairs help developers create high-quality final patches [40]. Because our DGA explicitly manages diversity in sub-populations, we hypothesize that it might produce a wider variety of distinct repairs. We measured repair 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 equal. With four participants, our DGA found 20% more unique repairs (harmonic mean over all benchmarks) than with one participant. If we consider only the difficult `flex`, `indent`, `merge` and `units` repairs, the improvement in the number of unique repairs found 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 [19, 36]).

There are several areas of potential future work. For example, we predict that variations of this technique could be used to optimize performance of programs for specific environments. It is known that seemingly innocuous machine-specific environmental factors can have surprisingly large impacts on the performance of common utility programs [28]. A second avenue for future work 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 for widely distributed programs to add diversity to deployed software.

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 repair algorithm shares this general weakness with all other approaches that use test suites or workloads to validate candidate repairs (e.g., Clearview [32] or GenProg [42]). 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 previous work 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 proposed *patch* representation [25] that supports a concise encoding of crossover.

7. SUMMARY AND CONCLUSION

This work extends previous techniques of automated program repair at the AST level to compiled (ASM) and linked (ELF) programs. The new representations enable program repair when source code cannot be parsed into ASTs (e.g., due to unavailable source files, complex build procedures or non-C source languages) and reduce memory and disk requirements sufficiently to make repair possible 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 which allows costly repair processes to be split across multiple devices.

Importantly for mobile and 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 by up to 62% (Section 4.4), which allows our approach to be applied to resource-constrained environments.

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 to propose 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 reduces fitness evaluation burdens by a factor of five — a super-linear 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 just under 900 bytes (or 7 SMS messages) sent 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 an important first step in the application of automated software repair to the growing field of mobile and embedded devices.

8. REFERENCES

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

[2] M. Barr. Faulty code will lead to an era of firmware-related litigation. In *Electronic Design*, Jan. 2010.

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

[4] 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*, Dec. 2009.

[5] 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.

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

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

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

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

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

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

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

[13] 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.

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

[15] 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.

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

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

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

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

[20] 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.

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

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

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

[24] 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.

[25] 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 (to appear)*, 2012.

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

[27] 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.

[28] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Architectural support for programming languages and operating systems*, pages 265–276, 2009.

[29] 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. MIT Press, June 1999.

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

[31] M. Orlov and M. Sipper. Genetic programming in the wild: evolving unrestricted bytecode. In *Genetic and evolutionary computation*, pages 1043–1050, 2009.

[32] 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.

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

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

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

[36] 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.

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

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

[39] 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.

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

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

[42] 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.