Deep Mutations have Little Impact

William B. Langdon and David Clark
W.Langdon@cs.ucl.ac.uk,david.clark@ucl.ac.uk
Department of Computer Science, University College London

ABSTRACT

Using MAGPIE (Machine Automated General Performance Improvement via Evolution of software), we measure the impact of genetic improvement (GI) on a non-deterministic deeply nested PARSEC VIPS parallel computing multi-threaded image processing benchmark written in C. More than 53% of mutants compile and generate identical results to the original program. We find about 10% Failed Disruption Propagation (FDP). Excluding internal errors and asserts, almost all changes deeper than 30 nested functions which are **E**xecuted and **I**nfect data or change control are not **P**ropagated to the output, i.e. these deep PIE changes have no external effect. Suggesting (where it relies on testing) automatic software engineering on deeply nested code will be hard.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**.

KEYWORDS

Software testing, robust software, fault masking, resilience, repair, automatic code optimisation, failed disruption propagation, FDP, PIE (propagation, infection, and execution), fitness landscape, information theory, genetic programming, local search, SBSE

ACM Reference Format:
William B. Langdon and David Clark. 2024. Deep Mutations have Little Impact. In *2024 ACM/IEEE International Workshop on Genetic Improvement (GI '24), April 16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3643692.3648259

1 INTRODUCTION

The robustness of software is a double edged sword. From the point of view of the user, having computer systems which do not fail is important, however from the perspective of software developers locating bugs in robust software and testing their fixes is hard. This slows down progress, forcing the user to deal with imperfect software which may have many defects or irritations which the development team in practice will never have time to resolve. Here we are primarily interested in automated software engineering, such as genetic improvement, but more robust, potentially more deeply nested programs, may be harder to repair, maintain and optimise either mechanically or manually.

GI '24, April 16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0573-1/24/04.
https://doi.org/10.1145/3643692.3648259

2 SOFTWARE ROBUSTNESS AND GENETIC IMPROVEMENT

By robust we mean that a system continues to operate even when perturbed. A robust system is still usable despite errors. If the perturbation is "small", a robust system will only deviate "slightly" or not at all from it usual behaviour and so remain usable. With larger perturbations a robust system may start to give larger responses. Only with very large perturbations will a robust system fail.

Globally we are now at the point where society relies on software, is even addicted to software [34], and although software is far from perfect[1], nonetheless it is used and delivers huge economic benefits [30][14]. Even though much effort is devoted to software verification and validation, particularly testing [17], including mutation testing [12, 21], in industry [20], society depends on buggy software, however real software is robust.

Previously [11, 40, 49] we found that software robustness can be explained by information theory [9, 44, 45] and an idea from software testing [58]. Voas and Miller [58] consider the difficulty of testing software, which can be considered as the other side of software robustness. They say for a software error to be seen the buggy code must be executed (their "E"), the execution must in some way change the internals of the program (they call this infection "I") and that the change must propagate ("P") to the program's output(s). Overall this is known as their "PIE" framework. "E", "I" and "P" must all be present for a code defect to impact the software. So, for example, if the bug lies in code which the genetic improvement (GI) fitness tests does not exercise (no "E") then the bug will have no fitness impact. If there is no measurable fitness impact, GI will find it very difficult to repair the bug.

We consider "P": does the disruption, if any, caused by the error propagate through the program to one or more of its outputs [1, 49]. If not, we call this failed disruption propagation (FDP). We [10] use information theory [9] to argue if there is information loss (measured by entropy loss, see Figure 1) on the route between the error (the infection point) and the program's output(s), then information about the error's disruption may be lost. If all information about the error is lost, then the program no longer depends on the error and so the error does not influence the output(s). Meaning the error does not have an externally measurable impact. That is, the software is robust to the error. We also suggest parts of a program may have more entropy loss, making the code before the high entropy loss region more robust. (In Section 6 we show this can happen in real programs, particular in deeply nested software.) Thus the effectiveness of genetic improvement depends not only on the error itself but do tests reach it (i.e. execute it), if so, does the test cause the bug to do something different (i.e. cause an infection) and *where* it is in the program, in terms of the test's *subsequent* path (execution trace) to the program's output(s).

[1]For example, Peng and Wallace [47, page v] said thirty years ago "errors will probably occur during software development and maintenance".

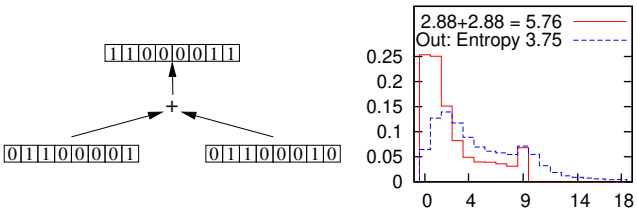

Figure 1: Left: adding two 8 bit numbers to give 8 bit result. Information is lost as inputs contain at most 2×8 bits (≤ 16 bits) of information and output can contain at most 8 bits. Right: red 0–9 actual distribution of 0-9 digits in 37 VIPS C source files. Dashed blue 0–18 distribution if they are added. Although the output of + is wider and has higher entropy (3.75), it is smoother and has less entropy than the combined entropy (5.76) of the two inputs to +. (Example expanded in Appendix A.)

In a strictly hierarchical system (see Figure 1), information only passes up through the hierarchy and once lost cannot be recovered. In terms of traditional genetic improvement (GI), if the disruption is lost before it reaches a measuring point (e.g. the program's output or a test oracle [32, 54]) there is no fitness signal and the GI has little chance of improving the code.

Niedermayr and Wagner [46] have already shown with Java mutation testing that there can be a strong relationship between the shortest path (their "minimal stack distance") from the test function (itself a Java method) to the mutated function and the effectiveness of the test. Notice, although they do not consider information theory, by using the shortest path they build in the assumption that test effectiveness falls with distance. Whereas we use the actual runtime nesting depth at which the mutation is executed[2]. In our C experiments there are no test methods, instead we use external test inputs and outputs and test the whole program. Thus, when we use the total nesting depth, it is akin to their stack distance but using the C `main` function instead of their Java test method. Also their JUnit tests have a maximum shortest path of 17 (average 8) [46, Fig. 3] whereas Magpie mutated VIPS code to a depth of up to 56 (Figures 4 and 5).

In low resolution systems we would expect more information loss. For example, in a system composed of only single bit logic gates, it may be difficult for disruption caused by an error to progress through many gates. For example, if the disruption signal encounters an And gate whose other argument is zero, then the gate's output is zero regardless of the disruption. That is, information about the error cannot propagate past the And gate. In general, the longer the path between the disruption and the GI's test point (test oracle) the more chance of entropy loss and so there is more chance that the disruption signal will not propagate.

In higher resolution system, e.g. 8 bit char (Figure 1) and 32 bit integer (which are the predominant types in our example, see Section 7.4), the information loss may be slower than in Boolean systems but, in general in hierarchical systems, it will occur. For example a multiplication operation (which scales the disruption signal) will destroy the signal if the multiplication's other argument is zero. Moreover any digital system is liable to lose information (only reversible computation does not lose information [24]). For example $x = a + b$ with $a = 2, b = 3$ and $a = 1, b = 4$ both set x to 5. That is, given the current value of x (5) we cannot infer the values of a and b. Note that, there was more information before the addition than afterwards. Even floating point arithmetic, which is designed to extract the maximum practical resolution from 32 bits, can lose information. For example, rounding error causes information loss [25]. With 32 bit IEEE floating point, $x = a + b$ with $a = 5.0, b = 0$ and $a = 5.0, b = 10^{-7}$ both set x to 5.0, so again information has been lost: from the output of the addition operation we cannot infer the values of its inputs.

Fitness landscape analysis is a relatively well studied topic in artificial evolution [41], however there is until now little work on the fitness landscape of real programs [48]. Some studies of C programs include [33, 39, 56, 57], where we enumerated the complete mutation landscape for the triangle program. In contrast Haraldsson et al. [18] used random walks to sample the fitness landscape for three fragments of python programs. Gabin An et al. [15] suggested, at least for PyGGI [16] and automated program repair, that AST mutations could be more effective than those acting directly on the source code. (Note here we use Magpie's AST mutations). While Smigielska et al. [53] analysed PyGGI [16] mutations for bug fixing on several Java QuixBugs programs.

Notice none of the above were interested in depth of nesting. Indeed researchers are usually interested in the size of programs rather than their depth [6, p15]. We did some work on integer [26] and floating point [28] functions, where fault masking could be total if the program nesting was deep enough, however all were artificially evolved (genetic programming [22, 50]) not real programs. For details see Section 7.4 in the discussion.

The next three sections describe how we use the Magpie GI tool (Section 3) to uniformly sample the space of mutations of the deeply nested VIPS C benchmark, including the fitness function (Section 4) and parameters (Section 5). Section 6 gives our results, including that only 17% of mutants fail at run time. Whilst Section 7 discusses Magpie on this bench mark. Finally we conclude (Section 8) that C software is robust to many AST based mutations and that failed disruption propagation (FDP) occurs more frequently with deeply nested mutants, making any form of test based automatic software engineering (such as genetic improvement) potentially more difficult in deeply nested code. Appendix A gives an information theory based explanation for FDP and mathematical formulae about it and entropy in special types of nested software.

3 MAGPIE

Magpie was initially released in 2022 as an open source project on GitHub[3]. As of 2 October 2023, including examples and documentation, Magpie contained 3781 lines of code, mostly written in Python. It contains examples in Python, C, C++ and Ruby.

[2]We use GNU libc `backtrace` to give depth of function nesting at run time.

[3]https://github.com/bloa/magpie 2 October 2023

3.1 PARSEC VIPS Benchmark

The VIPS image processing benchmark [43] is part of PARSEC (Princeton Application Repository for Shared-Memory Computers), which was devised as a benchmark to measure hardware performance on emerging workloads [2, page 73]. The PARSEC benchmark is often used, e.g. [7, 8, 13, 51, 52]. Indeed we used it in [35]. We downloaded the 64bit X86 version of PARSEC 3.0 from GitHub[4] and extracted the VIPS library from it. The VIPS thumbnail benchmark is often used but here our use is totally different. We do not want to automatically fix bugs but instead we use it as an example of highly nested well engineered software to demonstrate the effectiveness of Magpie's mutations and in particular how this varies with depth of procedural nesting in a multi-threaded parallel environment. Schulte et al. found significant improvements using their GOA [52]. GOA is a fitness driven evolutionary GI tool and so does not sample uniformly. As [52] does not report nesting depth, it may be that GOA found it easier to evolve the shallower parts of their VIPS.

3.2 Thirty Seven C Source files

We again use our VIPS C benchmark [35, Sect. 4.1]. VIPS is a large C library. Only a fraction of VIPS is used by each application. We took the VIPS thumbnail benchmark and instrumented it to select those source files which it uses on the test case (described in Section 4.1). Individual VIPS C source files were selected in two ways and then the union of the two taken. Firstly: the Linux perf tool was run at its maximum sampling frequency (40 kHz) ten times. All the functions perf profiled were included. Secondly: in all perf runs, the shrink_gen function stood out as consuming the most CPU time. Using the GDB debugger and setting a break point at shrink_gen the VIPS code was run multiple times and all the nested functions from `main` to shrink_gen were recorded. Despite non-deterministic multi-threading, this function nesting proved to be stable across multiple debugger sessions. Combining both approaches to find important functions lead to the identification of 37 source files. They also contain functions which are not used here. Automatically, at the individual function level, unused code was removed before presenting the source code to Magpie. Note this is only done to the function level. The C code to be mutated still contains some examples of if branch and `case` statements which are not used.

4 FITNESS FUNCTION

We are not attempting to improve the VIPS C code but to measure the impact of mutating it. Nevertheless we treat it as if we were running Magpie normally and supply it with a formal fitness function.

For each mutation we want to know:

(1) does it compile and link without error.
(2) does it run and terminate (within a two second time out[5]).
(3) does the program fail with an exception or error message.
(4) does the mutated program exit with a non-success exit status.
(5) does it generate an output and if so is the output mutated.

[4]https://github.com/bamos/parsec-benchmark/ 16 October 2023
[5]A unix `limit filesize` on the output was not needed.

Figure 2: 128×96 thumbnail image generated by VIPS.

4.1 Test Case

We used a GI benchmark PPM image (see Figure 2) [31, 35]. VIPS takes as input the 3264×2448 image (23 970 833 bytes) and generates a 128×96 PPM image as output (36 919 bytes) https://github.com/wblangdon/vips

5 MAGPIE SEARCH

Magpie has the ability to search using genetic programming [4] or local search [5] and to operate either in line mode or, as we used here, to treat the source files as AST trees. First the 37 C source files were converted to XML using scrml version 1.0.0. The ability to mutate and crossover XML gives Magpie the ability to work with any programming language. We sampled uniformly the impact of Magpie's seven common mutation operators 1000 times. Three operate at the C statement level: (StmtReplacement, StmtInsertion, StmtDeletion) and four make changes to parts of expressions (ComparisonOperatorSetting, ArithmeticOperatorSetting, NumericSetting, RelativeNumericSetting). For example, RelativeNumericSetting can change a value in the source code by 50%.

The Magpie parameter `max_steps` was set to one. Meaning each time Magpie created uniformly at random independently of execution depth of nesting a single mutant and tested it. For simplicity, `warmup` was left at 3, which means before each run, Magpie runs the non-mutated code three times. This is its standard noise suppression value[6]. Thus, Magpie created and tested the software four times for each mutation. The other Magpie parameters were left at their defaults.

Magpie used a mostly idle 32 GB 3.60 GHz Intel i7-4790 desktop CPU running networked Unix Centos 7, using Python 3 version 3.10.1 and version 10.2.1 of the GNU C compiler. On average, on a single 3.6GHz core, generating, compiling and testing each mutation takes, 2.453 seconds.

6 RESULTS

The results are summarised in Tables 1 and 2, whilst Figures 4 and 5 consider the variation of the impact of errors with stack depth.

Of the 1000 Magpie XML mutants, there are 302 which failed to compile (2^{nd} row in Table 1). These fall into 38 different classes. There are 177 compilation errors due to bad use of variable names, such as undeclared variables. The other 125 are essentially syntax errors. We discuss problems with moving variables out of their declaration scope in Section 7.3. It is surprising, given that Magpie

[6]It is possible to reduce (or increase) Magpie's pre-testing using its `[search] warmup = n` parameter (note $n \geq 1$).

Table 1: 1000 random Magpie VIPS mutants

Compiled, ran and produced correct output	526
Failed to compile	302
Failed to run correctly or gave incorrect output	164
Magpie TypeError[a]	8

[a] Magpie XML TypeError may have been fixed. GitHub commit b0ad2c1 (Oct 17, 2023)

Table 2: Details of Magpie 1000 VIPS mutants given in Table 1. Top two rows refer to the 526 successful mutants. Other seven are the 164 mutants which failed or gave bad output. Middle four rows mutants gave a non-success termination status.

Correct output	438
Mutation is identical to original code	88
Runtime error 134, e.g. assert, double free, mutex error	40
Exceed 2 second timeout	25
Segmentation error	22
Floating point error	4
VIPS detected error, e.g. No such file or directory	36
No error reported but output error	19
No error reported but output changed	18

is using XML and so is effectively operating at the program's AST level, that more than 12% of mutants fail to compile with syntax errors. Examples include pasting a well formed `if` statement into a `struct` data structure and replacing the minus sign in a negative constant (e.g. -1) with an arithmetic operator (e.g. /) giving rise to a syntax error (e.g. return /1;).

The last row in Table 1 says that there were 8 mutants where Magpie failed with an internal TypeError. It may be that these successfully passed the fitness tests. However it seems safest to exclude them. We also exclude the 88 identical mutants (second row in Table 2). So Tables 1 and 2 show 438 of 602 (1000-8-88-302), (i.e. 73%) of unique mutants which compile, produce the right output.

The middle four rows in Table 2 show 91 (55%) of the 164 mutants which compiled but gave bad results, failed whilst running. The last three rows in Table 2 show 73 (45%) of the erroneous mutants which ran either: VIPS detected an internal error (36 22%), the output was not generated (19 12%) or the image was created but was not the same as the original (18 11%). In six cases the output was the wrong size. But in 12 of the 18 cases where an incorrect output was generated the output was the right size. In some cases the incorrect output resembles the correct image (Figure 3) but in others although the image header in the output is correct, the image's content is totally scrambled. Notice Figure 3 indicates a different type of software robustness: although it is different from the correct output and thus fails the fitness test, visually it is "close" to the expected answer (Figure 2) and so might be acceptable.

6.1 Failed Disruption Propagation (FDP)

When considering failed disruption propagation in real code: disruptions to the program's internal state due to Magpie mutations which cause C exceptions or for which VIPS itself reports an error, are caught by special mechanisms which immediately terminate the program and so the disruption does not propagate through the program in the normal way (rows 3–7 in Table 2). The last two rows in Table 2 contain 37 mutations which either: caused the output not to be created or to be different in some way from the usual output. We uniformly at random selected 25 of these (blue cross hatch in Figures 4 and 5).

Figure 3: Almost all mutants which produce output, give images which are identical to Figure 2. Above is a similar but different mutant image.

From the mutants which did produce the right output (top row of Table 2), we uniformly at random selected 25 where: the modified code was executed and it changed the program's state or flow of control (shaded pink in Figures 4 and 5). (See also Table 3.) For both the selected 25 ok and 25 non-exception mutants (previous paragraph) we instrumented the mutation site to record how many times its execution made a difference and how deep in the function call hierarchy it was when it was executed.

The function containing the mutated code can be called multiple times and from different positions and hence the depth of a particular disruption typically varies during execution. (Perhaps due to use of multiple threads introducing non-determinism, there is sometimes a small variation between runs.) Although typically executed many times, in principle, only a single disruption need reach the output for the mutation to fail the test (Section 4) (blue hatching in Figures 4 and 5).

Note Figures 4 and 5 do not distinguish between levels of severity of the damage to the output. Either the mutant passed the test (pink) or it did not (blue hatch).

The histograms in Figure 4 are normalised so that if a mutation is executed and causes a change of state at different depth (plotted along the x-axes) the vertical height (y-axes) is plotted in proportion to the number of disrupting executions for that depth. This ensures that the area allocated to each of the (25+25) mutations plotted in Figure 4 is the same. Thus two mutations which both failed a test but one is executed many thousands of times and the other only once, are plotted in the same way. Disrupting executions of the same type (pass/failed) at the same depth are stacked on top of each other. For example in Figure 4, the peak (y=6) at depth x=8 represents all the failing disruptions at depth 8 across the 25 mutations randomly sampled from the 37 which failed without raising an error or exception[7].

The same data are presented in Figure 5, however the vertical (y) axis now represents the number of perturbations. That is, the y-axes shows the sum of all the disruptions of the same class (pass/fail)

[7] Actually 8 failing mutations introduce a disruption at depth 8. However 4 of these also cause disruptions at another depth. In this example, the 4 disrupt depth 8 exactly half the time, so giving $y = 6 = (4 + 4 \times \frac{1}{2})$ plotted in Figure 4.

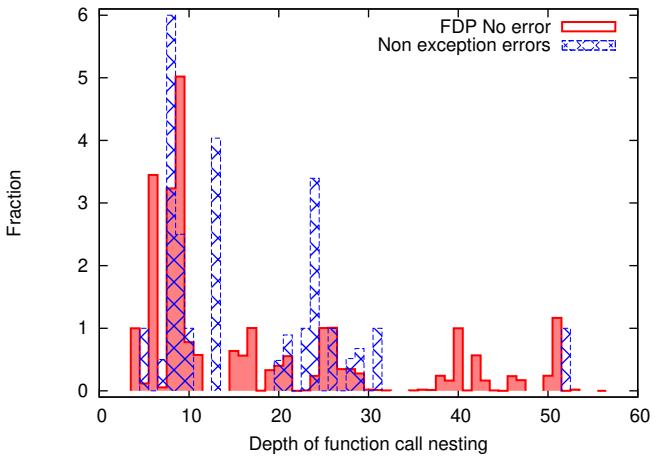

Figure 4: 25 mutations which change internal state but output is unaffected (shaded pink) and 25 which change output (pattern) without raising an exception or reporting an error. (See Section 6.1.)

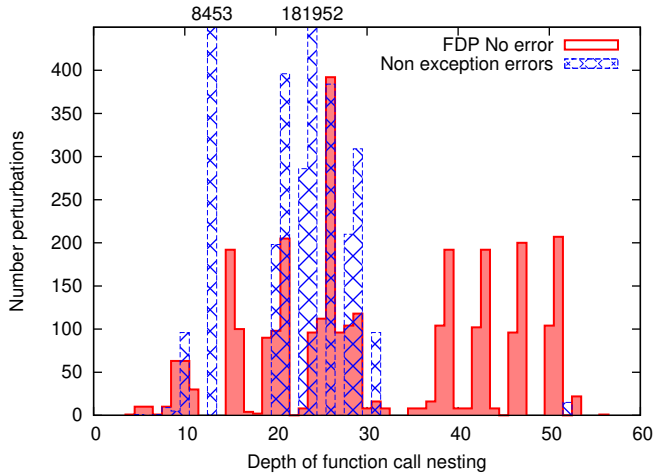

Figure 5: 25 mutations with no impact and 25 which change output. Same data as Figure 4. The vertical axis is truncated to 0–450, as otherwise perturbations which cause errors to the output (blue hatching) nested 13 functions deep x=13 (8453) and x=24 (181952), would dominate all the other data. (Graph described in Section 6.1.)

at the same depth (again disruptions which do reach the output are shown with blue hatching). Taking the example of the five failing mutations which change state at depth 24 (peak "181952" in Figure 5): two of them disrupt only at depth 24 (both infect 35 968 times); the other three disrupt at two or three depths (96, 96 and 109 824 at depth 24); total 181 952.

Notice failing mutations are typically executed causing disruptions more times and closer to the top of the stack (which in C means the `main()` function, depth 1). Whereas although disruptions which fail to propagate (FDP, pink shaded in Figure 5) can occur at a range of nesting levels, they predominate at depths greater than 30. I.e. there is more pink than blue right of x=30 in Figure 5.

Table 3: 91 random Magpie VIPS mutants without error. The first column says if the modified code is executed or not. "na" indicates that the mutant may or may not have been run, but in either case it cannot infect the state, e.g. replacing 0 by 0*3/2. 25 of 91 (27% ± 5%) mutants are executed and disrupt the program at least once.

Executed	Infected	count	fraction
N	N	45	49% ± 5%
na	N	13	14% ± 4%
y	N	8	9% ± 3%
y	y	25	27% ± 5%
		91	

We can estimate the fraction of FDP using data gathered from the non-error mutants when we sought our random sample of 25 mutants which did cause disruption but did not cause an error (see above). Table 3 considers 91 uniformly random chosen non-identical mutants of the 438 which run without error (first row Table 2). 25 of the 91 are executed and disrupt the program but do not change the output. This is 27% of the sample, which corresponds to 120 ± 21 in 438. In other words, for our VIPS about 12% ± 2% of Magpie mutants show failed disruption propagation (FDP). As mentioned in Section 3.2, in prior work [35] we already removed 2501 unused lines of C code (23%). Thus roughly in Table 3 we would expect the first two rows (corresponding to non-executed code) to increase by about 23%, and so the FDP rate for Magpie XML mutations to fall by about 23% to about 9% ± 1.6% in deeply nested C code. Similarly restoring the known to be unused code would increase the fraction of "neutral" mutations by about 23% to well over half, cf. top row of Table 1, making it still harder to make meaningful changes.

7 DISCUSSION

7.1 VIPS Typical of C Code?

Typically VIPS is composed of small functions which themselves are not deeply nested. Therefore it seems reasonable to use the depth of function nesting (i.e. position in the call stack) to serve as a proxy for the actual depth of nesting.

VIPS is typical of C programs. It uses both pointers to read and write data outside the current function and the function's own arguments and return value to pass information into and out of functions. That is, data and hence information flow is not tied exactly to control flow and the hierarchy of nested function calls. Nevertheless our results suggest in real code deeply nested functions can correspond to some extent to information loss regarding disruption caused by deeply buried errors.

7.2 Magpie identical Patches

The second row of Table 1 shows 9% of Magpie mutations are identical to the original code. It is therefore no surprise that they compile, run and generate identical output. Identical mutations are produced by XML operation ArithmeticOperatorSetting which only has 5 choices (+, −, ∗, /, %). So, for example, if the existing arithmetic operator is + there is a 1/5 chance that Magpie will replace + with another +, meaning no change is made. Whereas, although it is possible, the other XML changes are very unlikely

to replace the original XML with an identical copy. We observe 9% (rather than 1/5) because there are several other mutation operators as well as ArithmeticOperatorSetting (see Section 5).

It might be easy to force Magpie to ensure that new source code is different from the previous (parent) code. This seems like an obvious improvement, particularly for hill climbing local search. For population based search (i.e. genetic programming) these identical mutations represent a source of neutral moves [3, 51, 55], so removing them would change population dynamics, however it seems in general that removing them would not have a deleterious effect.

7.3 Magpie undeclared Variables

We saw in Section 6 that 18% of Magpie mutants fail to compile because the mutation has moved an existing variable out of scope. Usually a mutation failing to compile is a relatively cheap part of the fitness function. However the fraction of scope errors could potentially be addressed by:

- Restricting XML based mutations to copying source material within the same source file [38].
- Addition of new Magpie scope validity checks [37].
- Use SBSE [19] search techniques (such as genetic programming) to fix up variable names [42].

Moreover, as we did previously, e.g. [23], to further reduce the cost of erroneous mutations, we use the GCC command line option -fmax-errors=1 to stop the compiler immediately it discovers a single error.

7.4 VIPS few continuous types

Of the 1247 variables declared in our 37 C files, only 33 (2.6%) are continuous (float or double) or pointers to continuous variables. The other variables are discrete types (e.g. int, char, and application specific discrete types), indeed 69% are pointers to discrete variables. Like VIPS, many programs have few continuous data.

We would expect wide continuous data (e.g. 64 or 128 bit doubles) to be better at transmitting disturbances in information flow from one part of a program to another. It may be in some classes of program, which have many continuous variables, much deeper nesting will be needed to get the levels of fault masking seen here. However, albeit in a purely functional (Lisp) setting with 32 bit precision (float) we [27–29, 36] showed almost complete failure for sizeable disruptions to propagate to the output in very deep programs. Note we were concerned with functions evolved by genetic programming [22, 50], whereas here we deal with real programs written in C with data flows which do not slavishly follow the nested hierarchy of the procedure calls.

8 CONCLUSIONS

There are sound information theoretic reasons for expecting deeper software to be more robust (Section 2 and Appendix A). Although it is possible for information to flow through global shared data and so short circuit the hierarchy of nested functions, nonetheless we see a relationship between failed disruption propagation (FDP) and depth of program nesting in an imperative C program with a mixture of types (bytes, int, etc.) and scalars, pointers, arrays and compound data types (C `struct`).

Although it is disappointing to find 30% (Tables 1) of our mutations fail to compile, usually it is relatively cheap to detect compilation errors. (Sections 7.2 and 7.3 suggest potential improvements to Magpie.) If we exclude compilation errors, Magpie TypeError (Table 1), null mutations (Table 2) and take into account the unused functions (Section 3.2) about 90% of the time the code is robust to random code changes.

Most of the robustness is simply due to the mutation site not being executed (covered by tests). In perhaps another 9% or more, the mutant is exercised but has no impact (equivalent mutations are well known in mutation testing [59].) However we also find a significant proportion of FDP. Meaning, even if a perfect (possibly huge) test suite could be devised which covered the whole of the source code (i.e, "E" = 100%): the other two aspects of Voas' PIE framework [58] ("P" and "I") would still apply. If the new tests execute the code similarly to the existing tests, we would expect to find about (Table 3) 9% ± 3% of the time no change of state ("I"nfection) and 27% ± 5% FDP (incomplete "P"ropagation, total 36%), leading to the code passing the whole test suite.

Robust software continues operating despite errors, making it difficult to test and optimise. However, Schulte [51] and others have shown blackbox fitness driven optimisation can be applied to a wide range of programs. Nonetheless we suggest that it may be for larger, and especially deeper programs, far greater use of white box approaches with extensive internal instrumentation and closely packed and more sophisticated test oracles, will be needed by both automated testing and genetic improvement.

The fraction of failed disruption propagation (FDP) we see here is liable to be specific to the bench mark code and we would expected variation across applications and programming languages. Nonetheless we anticipate FDP, especially in deeper programs, is important to software robustness.

ACKNOWLEDGMENTS

I am grateful for the help of Aymeric Blot and the anonymous reviewers.

A LARGE EXPRESSIONS INFORMATION LOSS

Figures 6 and 7 expand the example in Figure 1 to a multiple level expression. As we proceed up the expression towards the output, as expected, we see information loss and so entropy falling. The expression has 6 inputs, each drawn from the same 0-9 distribution (the solid red line in Figure 7) and so (as in Section 2) each has entropy $H_o = 2.88$ bits. As the inputs are independent, the combined information content of the six digits is $6 \times 2.88 = 17.29$ bits. Adding two digits together gives a value in the range 0–18 (entropy 3.75, shown with dashed dark blue line in Figure 7). Although the blue line covers more values (0–18 v. 0–9) it is smoother than the original distribution (red line). The purple line shows the impact of multiplying the result of adding two digits by a third, giving values in the range (0-162, entropy 4.46). Notice further information loss indicated by total entropy falling again. Finally the light blue dotted line, the output, again shows the smoothing effect of addition and again total entropy falls (from $6 \times 2.88 = 17.29$ to 6.02 bits).

As we said in Section 2 all operators commonly used in programming lose information (i.e. are not reversible). In nested expressions

this loss is cumulative. So typically deeper expressions lose more information. In some special cases we can be precise.

In the case of the addition of n independent values, the mean is the sum of the individual means and similarly the variance is the sum of their variances. As n increases (by the central limit theorem) the output distribution will approach a Normal (Gaussian) distribution $N(m, \sigma^2)$. Where the mean is m and the standard deviation is σ (σ^2 = the variance). $N(m, \sigma^2)$ has entropy $\log_2(\sigma) + 2.0471$ bits.

In practice, assuming the individual distributions are not too different and not too asymmetric, the output distribution approaches $N(m, \sigma^2)$ rapidly (see Figures 8 and 10). Assume the inputs all come from the same distribution, with mean m_o and standard deviation σ_o. So mean $= n \times m_o$ and variance $\sigma^2 = n \times (\sigma_o)^2$, so $\sigma = \sqrt{n} \times \sigma_o$. Figure 8 plots the actual distributions for various numbers of independent inputs drawn at random from the distribution of 0-9 digits in the VIPS C source code used by Magpie. As expected, the mean and standard deviation follow $m = n \times m_o$ and $\sigma = \sqrt{n} \times \sigma_o$ (where $m_o = 2.53997$ and $\sigma_o = 2.75424$). The standard deviation is plotted with a dotted line in Figure 10 (note log scales)[8].

As n increases, then not only does the mean of $N(m, \sigma^2)$ increase but more importantly so to does its width σ. If we now consider that in a computer we are doing our calculations with a limited number of bits, so the infinite precision idealised $N(m, \sigma^2)$ has to be mapped into finite arithmetic. Suppose we use 8 bit integers, then then whole of $N(m, \sigma^2)$ is mapped onto 0–255 (see Figure 9). Regardless of the mean m, if the standard deviation σ is large compared to 255 then mapping the nicely curved distribution will lead

[8]A small animation of the output of expressions converging as they get bigger to the Gaussian distribution can be found on line via http://www.cs.ucl.ac.uk/staff/W.Langdon/icse2024/langdon_2024_GI/add10.html

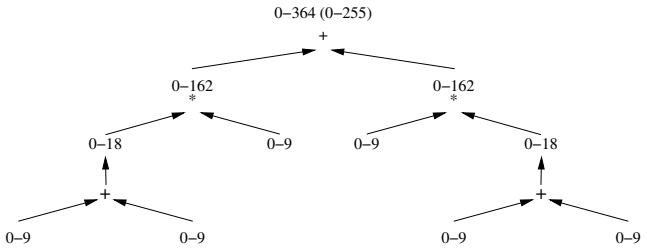

Figure 6: Example expression (a+b)c+d(e+f), of random (0-9)

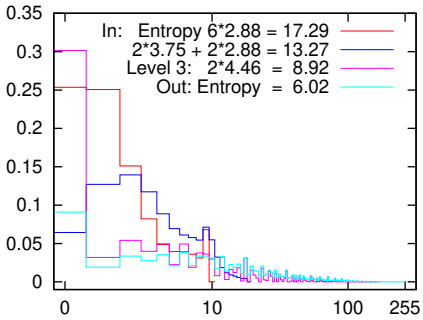

Figure 7: Distribution of values at each level in Figure 6. First two plots (red and blue) same as 2 plots in Figure 1. 8 bit precision, hence max entropy 8 and horizontal cut off at 255.

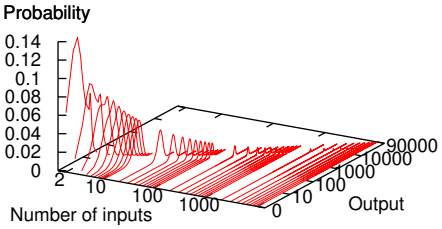

Figure 8: Distribution of values of adding random digits (0-9) from the VIPS source code. Note non-linear axis.

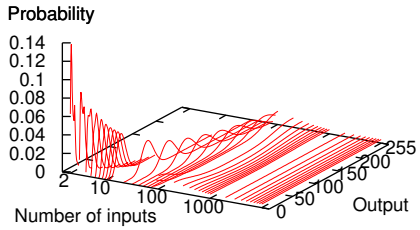

Figure 9: As Figure 8 but in 8 bits, hence cut off at 255

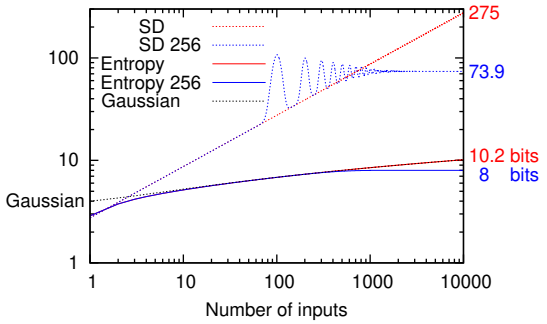

Figure 10: Standard deviation and entropy of adding VIPS digits 0 to 9. Note log-log plot. In red (upper traces of each pair) same data as Figure 8. Note convergence to Gaussian (lower dotted straight line) entropy $\log_2(\sigma) + 2.0471$. In blue same data as Figure 9, where sum is forced into 8 bit arithmetic and converges to uniform 0–255, entropy 8 bits.

to an almost uniform distribution across 0–255 (with an entropy of 8 bits). (Actually we get close, 3 significant digits, of a uniform distribution when σ is only 174.) Mathematically, if $\sqrt{n} \times \sigma_o \gg 256$ the entropy of the sum will be $\approx$ 8 bits and the information loss will be $\approx n\mathrm{H}_o - 8$ bits, i.e. almost all the input information is lost. Figure 10 shows for large n the theory agrees with actual values.

Finally: if the inputs to the expression are nicely behaved (meaning we can always take their logs) then the above argument can be extended to expressions with just multiplications. By taking logs, the expression changes from a series of multiplications of independent values to a sum of logs of independent values. Meaning we can again use the central limit theorem to argue that the sum will approach a Normal distribution, i.e. the product approaches a Log Normal distribution, with known information content as measured by entropy.

REFERENCES

[1] Kelly Androutsopoulos et al. 2014. An Analysis of the Relationship between Conditional Entropy and Failed Error Propagation in Software Testing. In *ICSE*. 573–583. http://dx.doi.org/10.1145/2568225.2568314

[2] Christian Bienia et al. 2008. The PARSEC benchmark suite. In *PACT*. 72–81. http://dx.doi.org/10.1145/1454115.1454128

[3] Aymeric Blot et al. 2015. Neutral but a Winner! How Neutrality Helps Multiobjective Local Search Algorithms. In *EMO (LNCS 9018)*. 34–47. http://dx.doi.org/10.1007/978-3-319-15934-8_3

[4] Aymeric Blot and Justyna Petke. 2020. Comparing Genetic Programming Approaches for Non-Functional Genetic Improvement Case Study: Improvement of MiniSAT's Running Time. In *EuroGP (LNCS 12101)*. 68–83. http://dx.doi.org/10.1007/978-3-030-44094-7_5

[5] Aymeric Blot and Justyna Petke. 2021. Empirical Comparison of Search Heuristics for Genetic Improvement of Software. *IEEE TEC* 25, 5 (2021), 1001–1011. http://dx.doi.org/10.1109/TEVC.2021.3070271

[6] Aymeric Blot and Justyna Petke. 2022. A Comprehensive Survey of Benchmarks for Automated Improvement of Software's Non-Functional Properties. arXiv. arXiv:2212.08540 [cs.SE] https://arxiv.org/abs/2212.08540

[7] Bobby R. Bruce et al. 2021. Enabling Reproducible and Agile Full-System Simulation. In *ISPASS*. 183–193. http://dx.doi.org/109/ISPASS51385.2021.00035

[8] Jie Chen and Guru Venkataramani. 2016. enDebug: A hardware-software framework for automated energy debugging. *JPDC* 96 (2016), 121–133. http://dx.doi.org/10.1016/j.jpdc.2016.05.005

[9] Rudi L. Cilibrasi and Paul M. B. Vitanyi. 2007. The Google Similarity Distance. *IEEE TKDE* 19, 3 (2007), 370–383. http://dx.doi.org/10.1109/TKDE.2007.48

[10] David Clark et al. 2020. Software Robustness: A Survey, a Theory, and Some Prospects. Presented at Facebook Testing and Verification Symposium 2020, 1-3 December.

[11] David Clark and Robert M. Hierons. 2012. Squeeziness: An information theoretic measure for avoiding fault masking. *Inf. Proc. Lett* 112, 8–9 (2012), 335–340. http://dx.doi.org/10.1016/J.IPL.2012.01.004

[12] Richard A. DeMillo et al. 1978. Hints on test data selection: Help for the practical programmer. *IEEE Computer* 11 (1978), 31–41. http://dx.doi.org/10.1109/C-M.1978.218136

[13] Jonathan Dorn et al. 2019. Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs. *IEEE TSE* 45, 3 (2019), 219–236. http://dx.doi.org/10.1109/TSE.2017.2775634

[14] Victoria A. Espinel. 2016. *The $1 Trillion Economic Impact of Software*. Technical Report. BSA, The Software Alliance. https://softwareimpact.bsa.org/pdf/Economic_Impact_of_Software_Report.pdf

[15] Gabin An et al. 2018. Comparing Line and AST Granularity Level for Program Repair using PyGGI. In *GI-2018*. 19–26. http://dx.doi.org/10.1145/3194810.3194814

[16] Gabin An et al. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *FSE*. 1100–1104. http://dx.doi.org/10.1145/3338906.3341184

[17] David Gelperin and Bill Hetzel. 1988. The Growth of Software Testing. *Comm. ACM* 31, 6 (1988), 687–695. http://dx.doi.org/10.1145/62959.62965

[18] Saemundur O. Haraldsson et al. 2017. Exploring Fitness and Edit Distance of Mutated Python Programs. In *EuroGP (LNCS 10196)*. 19–34. http://dx.doi.org/10.1007/978-3-319-55696-3_2

[19] M. Harman and B. F. Jones. 2001. Search Based Software Engineering. *Inf. Soft. Tech.* 43, 14 (2001), 833–839. http://dx.doi.org/10.1016/S0950-5849(01)00189-6

[20] Timo Hynninen et al. 2018. Software testing: Survey of the industry practices. In *MIPRO*. 1449–1454. http://dx.doi.org/10.23919/MIPRO.2018.8400261

[21] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE TSE* 37, 5 (2011), 649–678. https://doi.org/doi:10.1109/TSE.2010.62

[22] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. http://mitpress.mit.edu/books/genetic-programming

[23] W.B. Langdon and B.J. Alexander. 2023. Genetic Improvement of OLC and H3 with Magpie. In *GI@ICSE*. 9–16. http://dx.doi.org/10.1109/GI59320.2023.00011

[24] W. B. Langdon. 2003. The distribution of Reversible Functions is Normal. In *GPTP*. Kluwer, Chapter 11, 173–187. http://dx.doi.org/10.1007/978-1-4419-8983-3_11

[25] William B. Langdon. 2022. Dissipative Arithmetic. *Complex Systems* 31, 3 (2022), 287–309. http://dx.doi.org/10.25088/ComplexSystems.31.3.287

[26] William B. Langdon. 2022. Failed Disruption Propagation in Integer Genetic Programming. In *GECCO-2022 Companion*. 574–577. http://dx.doi.org/10.1145/3520304.3528878

[27] W. B. Langdon. 2022. Genetic Programming Convergence. *GP & EM* 23, 1 (2022), 71–104. http://dx.doi.org/10.1007/s10710-021-09405-9

[28] W. B. Langdon. 2022. Open to Evolve Embodied Intelligence. In *EI-2022*. IOP Publishing. http://dx.doi.org/10.1088/1757-899X/1292/1/012021

[29] W. B. Langdon. 2022. A Trillion Genetic Programming Instructions per Second. ArXiv. https://arxiv.org/abs/2205.03251

[30] W. B. Langdon. 2023. The End Is Not Clear. *Comm. ACM* 66, 7 (2023), 9. http://dx.doi.org/10.1145/3596710

[31] William B. Langdon et al. 2016. API-Constrained Genetic Improvement. In *SSBSE (LNCS 9962)*. 224–230. http://dx.doi.org/10.1007/978-3-319-47106-8_16

[32] William B. Langdon et al. 2017. Inferring Automatic Test Oracles. In *SBST*. 5–6. http://dx.doi.org/10.1109/SBST.2017.1

[33] William B. Langdon et al. 2017. Visualising the Search Landscape of the Triangle Program. In *EuroGP*. 96–113. http://dx.doi.org/10.1007/978-3-319-55696-3_7

[34] William B. Langdon et al. 2021. Information Loss Leads to Robustness. IEEE Software Blog. http://blog.ieeesoftware.org/2021/09/information-loss-leads-to-robustness-w.html

[35] William B. Langdon et al. 2024. Genetic Improvement of Last Level Cache. In *EuroGP (LNCS)*.

[36] William B. Langdon and Wolfgang Banzhaf. 2022. Long-Term Evolution Experiment with Genetic Programming. *Artificial Life* 28, 2 (2022), 173–204. http://dx.doi.org/10.1162/artl_a_00360

[37] William B. Langdon and Mark Harman. 2014. Genetically Improved CUDA C++ Software. In *EuroGP (LNCS 8599)*. 87–99. http://dx.doi.org/10.1007/978-3-662-44303-3_8

[38] William B. Langdon and Mark Harman. 2015. Optimising Existing Software with Genetic Programming. *IEEE TEC* 19, 1 (2015), 118–135. http://dx.doi.org/10.1109/TEVC.2013.2281544

[39] William B. Langdon and Mark Harman. 2016. Fitness Landscape of the Triangle Program. In *PPSN-2016 Workshop on Landscape-Aware Heuristic Search*. http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/rn1605.pdf

[40] William B. Langdon and Justyna Petke. 2015. Software is Not Fragile. In *CS-DC'15*. 203–211. http://dx.doi.org/10.1007/978-3-319-45901-1_24

[41] Katherine Mary Malan. 2021. A Survey of Advances in Landscape Analysis for Optimisation. *Algorithms* 14 (2021), 40. http://dx.doi.org/10.3390/a14020040

[42] Alexandru Marginean et al. 2015. Automated Transplantation of Call Graph and Layout Features into Kate. In *SSBSE (LNCS 9275)*. 262–268. http://dx.doi.org/10.1007/978-3-319-22183-0_21

[43] K. Martinez and J. Cupitt. 2005. VIPS a highly tuned image processing software architecture. In *ICIP*. 574–577. http://dx.doi.org/10.1109/ICIP.2005.1530120

[44] Ibrahim Mesecan et al. 2021. CRNRepair: Automated Program Repair of Chemical Reaction Networks. In *GI@ICSE*. 23–30. http://dx.doi.org/10.1109/GI52543.2021.00014 Winner Best Paper.

[45] Ibrahim Mesecan et al. 2021. HyperGI: Automated Detection and Repair of Information Flow Leakage. In *ASE NIER track*. 1358–1362. arXiv:2108.12075 http://dx.doi.org/10.1109/ASE51524.2021.9678758

[46] Rainer Niedermayr and Stefan Wagner. 2019. Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?. In *EASE*. 189–198. http://dx.doi.org/10.1145/3319008.3319021

[47] Wendy W. Peng and Dolores R. Wallace. 1993. *Software Error Analysis*. NIST Special Publication 500-209. Computer Systems Technology, U.S. Department of Commerce. Technology Administration National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-209.pdf

[48] Justyna Petke et al. 2019. A Survey of Genetic Improvement Search Spaces. In *GI@GECCO*. 1715–1721. http://dx.doi.org/10.1145/3319619.3326870

[49] Justyna Petke et al. 2021. Software Robustness: A Survey, a Theory, and Some Prospects. In *FSE IVR*. 1475–1478. http://dx.doi.org/10.1145/3468264.3473133

[50] Riccardo Poli et al. 2008. *A field guide to genetic programming*. http://www.gp-field-guide.org.uk (With contributions by J. R. Koza).

[51] Eric Schulte. 2014. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution*. Ph. D. Dissertation. University of New Mexico. http://hdl.handle.net/1928/25819

[52] Eric Schulte et al. 2014. Post-compiler Software Optimization for Reducing Energy. In *ASPLOS*. 639–652. http://dx.doi.org/10.1145/2541940.2541980

[53] Marta Smigielska et al. 2021. Uniform Edit Selection for Genetic Improvement: Empirical Analysis of Mutation Operator Efficacy. In *GI@ICSE*. 1–8. http://dx.doi.org/10.1109/GI52543.2021.00009

[54] Valerio Terragni et al. 2020. Evolutionary Improvement of Assertion Oracles. In *FSE*. 1178–1189. http://dx.doi.org/10.1145/3368089.3409758

[55] Ting Hu et al. 2020. A network perspective on genotype-phenotype mapping in genetic programming. *GP & EM* 21, 3 (2020), 375–397. http://dx.doi.org/10.1007/s10710-020-09379-0

[56] Nadarajen Veerapen et al. 2017. Modelling Genetic Improvement Landscapes with Local Optima Networks. In *GI-2017*. 1543–1548. http://dx.doi.org/10.1145/3067695.3082518 Best Presentation prize.

[57] Nadarajen Veerapen and Gabriela Ochoa. 2018. Visualising the global structure of search landscapes: genetic improvement as a case study. *GP & EM* 19, 3 (2018), 317–349. http://dx.doi.org/10.1007/s10710-018-9328-1

[58] Jeffrey M. Voas and Keith W. Miller. 1995. Software Testability: The New Verification. *IEEE Software* 12, 3 (1995), 17–28. http://dx.doi.org/10.1109/52.382180

[59] Xiangjuan Yao et al. 2014. A Study of Equivalent and Stubborn Mutation Operators using Human Analysis of Equivalence. In *ICSE*. 919–930. http://dx.doi.org/10.1145/2568225.2568265