Automated Software Engineering
doi:10.1007/s10515-024-00475-4

Deep Imperative Mutations have Less Impact

W. B. Langdon and David Clark

16 November 2024

Abstract Information theory and entropy loss predict deeper more hierarchical software will be more robust. Suggesting silent errors and equivalent mutations will be more common in deeper code, highly structured code will be hard to test, so explaining best practise preference for unit testing of small methods rather than system wide analysis. Using the genetic improvement (GI) tool MAGPIE, we measure the impact of source code mutations and how this varies with execution depth in two diverse multi-level nested software. gem5 is a million line single threaded state-of-the-art C++ discrete time VLSI circuit simulator, whilst PARSEC VIPS is a non-deterministic parallel computing multi-threaded image processing benchmark written in C. More than 28–53% of mutants compile and generate identical results to the original program. We observe 12% and 16% Failed Disruption Propagation (FDP). Excluding internal errors, exceptions and asserts, here most faults below about 30 nested function levels which are **E**xecuted and **I**nfect data or divert control flow are not **P**ropagated to the output, i.e. these deep PIE changes have no visible external effect. Suggesting automatic software engineering on highly structured code will be hard.

Keywords: Software testing, genetic improvement (GI), robust software, fault masking, fault localization, resilience, repair, automatic code optimisation, failed disruption propagation, FDP, Voas PIE (propagation, infection, and execution), fitness landscape, information theory, genetic programming, genetic improvement, local search, SBSE, image processing, RNAfold

1 Introduction

The robustness of software, Petke et al (2021), is a double edged sword. From the point of view of the user, having computer systems which do not fail is

University College, London, Gower Street, London, WC1E 6BT, UK

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.

2 Software Robustness and Genetic Improvement

By robust we mean that a system continues to operate even when perturbed[1]. 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 deviations from its normal behaviour. 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 Langdon et al (2021), and although software is far from perfect[2], nonetheless it is used and delivers huge economic benefits Langdon (2023), Espinel (2016). Even though much effort is devoted to software verification and validation, particularly testing Gelperin and Hetzel (1988), including mutation testing DeMillo et al (1978); Jia and Harman (2011); Xiangjuan Yao et al (2014), in industry Hynninen et al (2018), society depends on buggy software, however real software is robust.

Previously Petke et al (2021), Clark and Hierons (2012), Langdon and Petke (2015) we found that software robustness can be explained by information theory Cilibrasi and Vitanyi (2007), Mesecan et al (2021b), Mesecan et al (2021a) and an idea from software testing. Voas and Miller (1995) 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.

[1] We follow Petke et al (2021) and consider perturbations of all sorts from normal behaviour. A perturbation may be long lasting or transient. For example, it may be due to a bug, coding error, software mutation, power spike, cosmic radiation or malicious actor.

[2] For example, (Peng and Wallace 1993, page v) said thirty years ago "errors will probably occur during software development and maintenance".

We consider "P": does the disruption, if any, caused by the error propagate through the program to one or more of its outputs Petke et al (2021); Androutsopoulos et al (2014). If not, we call this failed disruption propagation (FDP). We use information theory 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 Clark et al (2020). 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).

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 Terragni et al (2020); Langdon et al (2017a)) there is no fitness signal and the GI has little chance of improving the code.

Niedermayr and Wagner (2019) 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. In our C/C++ experiments there are no test methods, instead we use external test inputs and outputs and test the whole program (system tests). Thus, when we use the total nesting depth[3], it is akin to their stack distance but using the C/C++ `main` function instead of their Java test method. Also their JUnit tests have a maximum shortest path of 17 (average 8) (Niedermayr and Wagner 2019, Fig. 3) whereas Magpie mutated VIPS code to a depth of up to 56 (Figures 7 to 9) and gem5 up to 85 (Figures 10 to 12).

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. Suppose the disruption signal encounters an And gate whose other input is false, then the gate's output is false 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

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

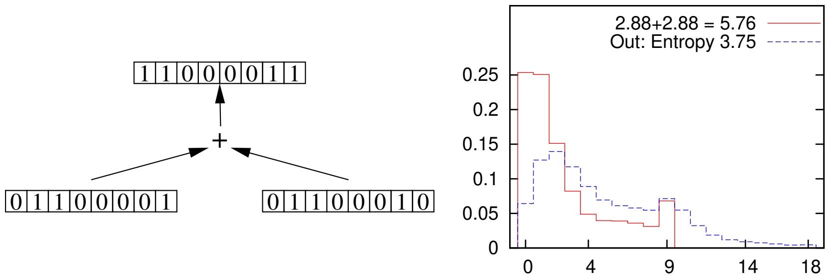

Fig. 1 Left: adding two 8 bit numbers to give 8 bit result. Information is lost as inputs contain at most 2×8 bits (≤ 16 bits) of information and output can contain at most 8 bits. Right: red 0–9 actual distribution of 0-9 digits in 37 VIPS C source files. Dashed blue 0–18 distribution if they are added. 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 the appendix.)

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 examples, 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 Langdon (2003)). 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 Langdon (2022a). 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 Malan (2021), however there is until now little work on the fitness landscape of real programs Petke et al (2019). Some studies of C programs include Langdon and Harman (2016), Langdon et al (2017b), Veerapen et al (2017), Veerapen and Ochoa (2018), where we enumerated the complete mutation landscape for the triangle program. In contrast Haraldsson et al (2017) used random walks to sample the fitness landscape for three fragments of python programs. Gabin An et al (2018) suggested, at least for automated program repair using PyGGI Gabin An et al (2019), that AST mutations could be more effective than mutating source code directly (note we use Magpie's AST mutations, Section 3.1). While Smigielska et al (2021) analysed PyGGI 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 (Blot and Petke 2022a, p15). We did some work on integer Langdon (2022b) and floating point Langdon (2022d) functions, where fault masking could be total if the program nesting was deep enough, however all were artificially evolved (genetic programming Koza (1992), Poli et al (2008)) 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 and C++ gem5, including the fitness function (Section 4) and parameters (Section 5). Section 6 gives our results, including that only 17% of VIPS and 22% of gem5 mutants fail at run time. Whilst Section 7 discusses Magpie on our examples, including examples of the mechanisms behind FDP (Section 7.5). Finally we conclude (Section 8) that C/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. The appendix gives an information theory based explanation for FDP and mathematical formulae about it and entropy in special types of nested software.

3 Experiments to Study Disruption Propagation in C/C++

3.1 Magpie for Mutation Sampling and Impact Measurement

Magpie is a language independent genetic improvement tool. We use to it to generate uniformly at random mutations and measure their impact. Magpie was initially released in 2022 as an open source project on GitHub[4]. As of 2 October 2023, including examples and documentation, Magpie contained 3781 lines of code, mostly written in Python. It contains worked examples in Python, C, C++ and Ruby. The next two sections describe VIPS (3.2 and 3.3) while the following two refer to gem5 (3.4 and 3.5).

3.2 PARSEC VIPS Benchmark

The VIPS image processing benchmark Martinez and Cupitt (2005) is part of PARSEC (Princeton Application Repository for Shared-Memory Computers), which was devised as a benchmark to measure hardware performance on emerging workloads (Bienia et al 2008, page 73). The PARSEC benchmark is often used, e.g. Schulte et al (2014), Schulte (2014), Chen and Venkataramani (2016), Dorn et al (2019), Bruce et al (2021). Indeed we used it in Langdon and Clark (2024b). We downloaded the 64bit X86 version of PARSEC 3.0

[4] `https://github.com/bloa/magpie` 2 October 2023 Blot and Petke (2022b)

from GitHub[5] 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 Schulte et al (2014). GOA is a fitness driven evolutionary GI tool and so does not sample uniformly. As Schulte et al (2014) do not report nesting depth, it may be that GOA found it easier to evolve the shallower parts of their VIPS.

3.3 VIPS Thirty Seven C Source files

We again use our VIPS C benchmark (Langdon and Clark 2024b, 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 VIPS C code to be mutated still contains some examples of if branch and `case` statements which are not used.

3.4 gem5 Benchmark

gem5 Binkert et al (2011) Bruce et al (2021) is the state of the art simulation tool for systems composed of very large scale integrated (VLSI) electronic circuits. It is widely used by industrial chip designers and manufactures and for open source and academic research. It supports most commercial CPU instruction sets (ISAs) and popular memory architectures. gem5 is an open project available via GitHub. It was written and has been maintained for more than 10 years by a team of expert C++ programmers.

[5] `https://github.com/bamos/parsec-benchmark/` 16 October 2023

For SSBSE 2023 Arcaini et al (2023), Bobby Bruce cloned gem5 staging branch v23.0, included the latest features and improvements and ssbse-challenge-examples and merged them into a stable release. As part of the SSBSE 2023 challenge Dakhama et al (2023), we cloned the SSBSE version of gem5[6]. It comprises a total of 1.34 million lines of code (mostly C++) (git commit: 65edbe0, Jul 14, 2023).

gem5 is a complete discrete time simulation and typically runs of the order of 10^5 times slower than the circuit it is simulating. (For example, with our RNAfold fragment, Section 4.2, gem5 runs 108 000 times slower than real time.) Thus to simulate 1000 clock ticks on a 3.6GHz CPU will take about 30 milliseconds.

3.5 gem5 Twenty Five C++ Source Files

As mentioned in Section 3.4, gem5 is a huge program. Starting from its almost 10 000 source files, on a single core, it takes more than two hours to compile and build gem5 to target only X86 binaries. Therefore gem5 was profiled using GNU gcov on our test case (Section 4.2) and 25 heavily used C++ source files were selected to be used by Magpie (see also Figure 2). Instead of the gem5 scons build script, a conventional Linux command script was written to compile just the mutated code and link it against the gem5 shared object library. For compilable mutants, compiling and linking takes on average 7 seconds. Notice for gem5, unlike VIPS, we did not seek to exclude unused code. Instead we reject mutations which according to gcov line coverage profile are not used on the test case. This leads to rejecting 69.2% of gem5 Magpie mutations before they are compiled (Table 4). This has the advantage that the gem5 C++ source files do not need to be stripped of their unused functions and the gcov profile says which lines of code are used during fitness testing (rather than which functions are called). Since we know where Magpie has placed its mutation, it is easy to use the pre-collected profile data to quickly weed out useless mutations, rather than run the complete fitness evaluation to simply confirm it has no runtime influence (as it is not executed).

4 Fitness Function

We are not attempting to improve VIPS or gem5 but to measure the impact of mutating their C/C++ sources. 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 time limit (VIPS 2, gem5 15 seconds)[7].

[6] gem5 `https://github.com/BobbyRBruce/gem5-ssbse-challenge-2023.git`
[7] A unix `limit filesize` on the output was not needed.

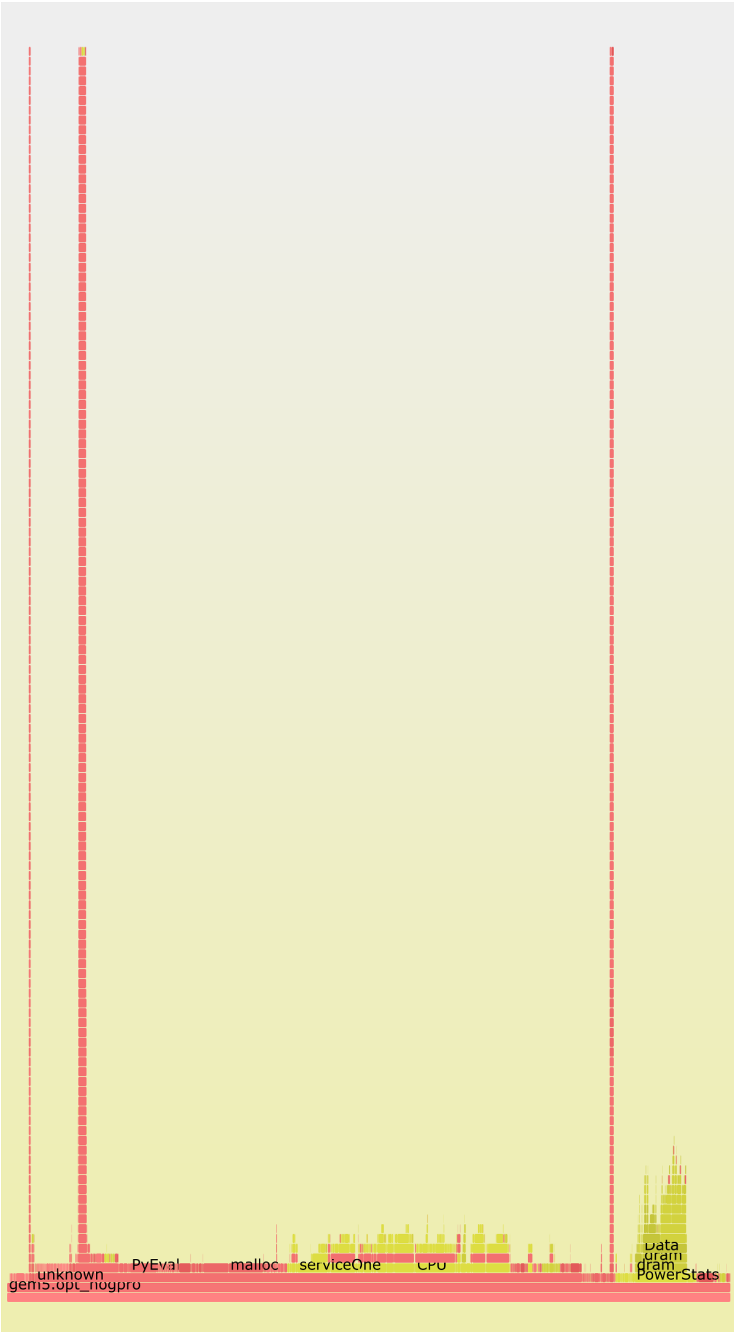

Fig. 2 FlameGraph of Linux perf profile of gem5 simulating our RNAfold fragment (Section 4.2). Used functions are spread horizontally, whilst vertical axis indicates depth of function call nesting. (An interactive version is available via `https://github.com/wblangdon/Deep-Imperative-Mutations-have-Less-Impact`)

Fig. 3 Left: VIPS 3264×2448 benchmark input image (23 970 833 bytes) Right: 128×96 thumbnail image generated by VIPS (36 919 bytes, see left of Figure 5 for enlarged thumbnail).

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.

4.1 VIPS Test Case

We used a GI benchmark PPM image (see Figure 3) Langdon et al (2016), Langdon and Clark (2024b). VIPS takes as input the 3264×2448 image (23 970 833 bytes) and generates a 128×96 PPM image as output (36 919 bytes)[8].

4.2 gem5 Test Case

gem5 is used to simulate a CPU intensive loop written in C and running on a 64 bit X86 computer (gem5 command line option `--isa X86`). We used the default configuration script supplied with SSBSE 2023 challenge track (gem5 command line input `hello-custom-binary.py`). For the X86 program that gem5 simulates we took the most compute intensive loop from the open source RNAfold[9] program (gem5 command line option `--binary higher_order_code_209`). Otherwise we used gem5's defaults, including disabling debug options.

[8] VIPS benchmark `https://github.com/wblangdon/vips`

[9] RNAfold Lorenz et al (2011) calculates the minimum self-binding free energy of an RNA molecule. It is written in C and is part of the open source ViennaRNA package `https://www.tbi.univie.ac.at/RNA/`

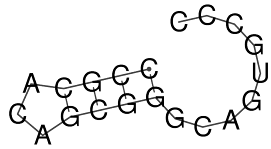

Fig. 4 Twenty base RNA molecule used in gem5 test case `higher_order_code_209`. The figure shows the minimum free energy secondary structure, which is found by RNAfold. Note the C to G pair bindings form a characteristic low energy "hairpin" spiral, often found in both RNA and DNA molecules.

Like RNAfold version 2.5.1 itself, the X86 executable `higher_order_code_209` was compiled with gcc -O2. It repeats the 209 iterations of the loop needed for an example twenty base RNA molecule, Figure 4. (RNAfold runtime grows faster than quadratically with RNA molecule size, hence a small RNA molecule was used.) We had previously used Genetic Improvement to improve RNAfold's accuracy Langdon et al (2018) and to parallelise this loop Langdon and Lorenz (2017, 2019).

5 Magpie Search

Magpie has the ability to search using genetic programming Blot and Petke (2020) or local search Blot and Petke (2021) and to operate either in line mode or, as we do here, to treat the source files as AST trees. First the 37 VIPS C (Section 3.3) and 25 gem5 C++ (Section 3.5) source files were converted to XML files 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 at random the impact of Magpie's seven common mutation operators 1000 times (gem5 2500[10]). Three mutate C/C++ statements (StmtReplacement, StmtInsertion, StmtDeletion) whilst four change parts of expressions within statements (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. The other Magpie parameters were left at their defaults.

Magpie used a mostly idle 32 GB eight core 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/C++ compiler. On average generating compiling and testing each VIPS mutation takes 2.5 seconds. Whilst for gem5 it is

[10] In Section 3.5 we noted the higher fraction of gem5 mutations falling in non-executed code and in order to get at least the 37 non-exception runtime errors we found with VIPS Langdon and Clark (2024a), Table 2, we increase our gem5 sample size to 2500. Actually Table 3 reports we found 55 gem5 mutants which gave at least one wrong answer at runtime but did not raise an exception. (We need at least 25 for the comparison in Section 6.4.)

6.6 second. Of course gem5 is a much bigger program, 44MB v. 450KB, and for example, linking gem5 alone takes on average about 1.4 seconds, whilst running takes on average about 1.1 seconds v. about 80 milliseconds for VIPS (albeit VIPS uses all 8 available cores).

6 Results

The next two sections give the fraction of VIPS (6.1) and gem5 (6.2) mutations in each impact class, before Sections 6.3 and 6.4 consider in detail the variation of the impact of mutations with run time depth.

6.1 VIPS Results

The VIPS results are summarised in Tables 1, 2, and 5 whilst Figures 7 to 9 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 7.2. It is surprising, given that Magpie 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 VIPS 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 with an exception 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. (The left side of Figure 5 shows the correct output v. error on right.) In others although the image header in the output is correct, the image's content is totally scrambled (Figure 6). Notice Figure 5 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 and so might be acceptable.

Table 1 1000 random Magpie VIPS mutants

Compiled, ran and produced correct output	526	52.6 %
Failed to compile	302	30.2 %
Failed to run correctly or gave incorrect output	164	16.4 %
Magpie TypeError[a]	8	0.8 %

[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 (Section 7.1 on row 2). 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	43.8 %
Mutation is identical to original code	88	8.8 %
Runtime error 134, e.g. assert, double free, mutex error	40	4.0 %
Exceed 2 second timeout	25	2.5 %
Segmentation error	22	2.2 %
Floating point error	4	0.4 %
VIPS detected error, e.g. No such file or directory	36	3.6 %
No error reported but output error	19	1.9 %
No error reported but output changed	18	1.8 %

Fig. 5 Left: original VIPS thumbnail output. Almost all mutants which produce output, give images which are identical. Right: a similar but different mutant image.

Fig. 6 Note most mutant images are unchanged (Figure 5), however above is a radically different mutant image. Note although the pixels are scrambled, the output is still an image and of the right type and dimensions.

6.2 gem5 Results

The gem5 results are summarised in Tables 3, 4 and 5 whilst Figures 10 to 12 consider the variation of the impact of errors with stack depth. To allow easy comparison between gem5 and VIPS results Tables 3 to 5 and Figures 10 to 12 follow the same format as the VIPS results in the previous section.

Of the 2500 Magpie gem5 mutants, most (1730, Table 4) are rejected because they lie in non-executed code (see Section 3.5 page 7).

The second row of Table 4 shows 238 gem5 mutants are rejected because actually Magpie made no change. E.g. because a number mutation replaced 0 with another 0. This is more-or-less the same ratio (9.52%) as VIPS 8.8% (Table 2).

A further 17 Magpie mutations are rejected because, although syntactically different (when compiled with -O2) their object code is identical. For example, a mutation which inserted "`Tick when = 0;`" where the compiler recognises that the variable "`when`" is unused and optimises it away, leaving the rest of the object code unchanged. We have previously used this with the GCC compiler Langdon (2020) and LLVM Langdon et al (2023) to quickly spot semantically identical mutations and so avoid the cost of fitness evaluation. It can also be used with the compiler's assembler code output to spot semantically identical parts of mutations when automatically simplifying compound mutations Langdon (2020). The compiler C pre-processor can also be used to strip away parts of mutations rendered irrelevant by conditional compilation directives when building tabu list of mutations Langdon et al (2015). Just running the pre-processor is typically much cheaper than running the complete compilation. The compiler has been used to identify equivalent mutants in mutation testing Papadakis et al (2015).

Of the 532 (2500-1730-238) gem5 compilations, 228 (43%) mutants failed to compile (Table 4). These fall into 15 different classes. There are 164 (31%) compilation errors due to bad use of variable names (scoping errors will be discussed in Section 7.2). The other 64 (12%) compilation errors are different types of syntax error. Syntax error include removing the `if` from an `if else` leaving the `else` dangling and replacing a `*` used to dereference a pointer with an arithmetic operator, such as `-`.

The middle four rows in Table 4 show 77 (53%) of the 145 mutants which compiled but gave bad results, failed with a system exception whilst running. A further 13 failed with one of gem5's exceptions (total 62%). For example, one of Magpie's mutations changed the condition in a `while` loop so that it was always false, meaning the size etc. of a buffer was not set up. This later resulting in gem5 detecting a `fatal` exception in `writeBlob` and so it stopped with status code 1 and much of the gem5 output was not created.

The last but one row of Table 4 shows 55 mutations compiled and ran without reporting an error but gave erroneous outputs (38% of the 145 mutations which failed at runtime). For example, in one case Magpie mutated the initial start condition in a for loop from `i = 0` to `i = -1`, resulting in the loop starting with an illegal value for variable `i`. Notice the mutated for loop was

Table 3 2500 random Magpie gem5 mutants

Compiled, ran and produced correct output	380	15.2 %
Failed to compile etc.	1975	79.0 %
Failed to run correctly or gave incorrect output	145	5.8 %
Magpie TypeError	0	0 %

Table 4 Details of Magpie 2500 gem mutants given in Table 3. Top row refers to the 142 successful mutants. The next three rows the mutation was ok but rejected before testing due to: no change to source code (238), object files are identical (17) or it was located in code that profiling said is not executed on the test case (1730). Last column gives percentages excluding these automatically rejected equivalent mutations. 228 mutations failed to compile. The other six rows are the 145 mutants which failed or gave bad output. Middle four rows mutants gave a non-success termination status.

Correct output	142	5.68 %	27.57 %
Mutation is identical	238	9.52 %	-
Mutation is semantically identical	17	0.68 %	-
Mutation in non-executed code	1730	69.20 %	-
Failed to compile	228	9.12 %	44.27 %
gem5 detected error, e.g. gem5 panic, Assertion failure or it erronously reports “segmentation fault” in code it is simulating.	47	1.88 %	9.13 %
Exceed 15 second timeout	17	0.68 %	3.30 %
Segmentation error	10	0.40 %	1.94 %
Floating point error	3	0.12 %	0.58 %
gem5 detected “fatal:” error	13	0.52 %	2.52 %
No error reported but output changed	55	2.20 %	10.68 %
Totals	2500		(515)

executed for one more iteration than it should have been. gem5 did not notice the error. Four of the five output files created by gem5 were unchanged. Only 8 of the 504 lines in the other file were changed. Indeed all numeric values in it were unchanged and the only change was that in the 8 cases the text description at the start of the line was slightly corrupted.

6.3 VIPS 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 7 and 8).

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 7 and 8). (See also Figure 9 and left of Table 5.) 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 the use of multiple threads introducing non-determinism, there is sometimes a small variation between runs.) Although typically executed many times only a single disruption need reach the output for the mutation to fail the test (Section 4) (blue hatching in Figures 7 and 8).

Note Figures 7 to 9 do not distinguish between levels of severity of the damage to the output. Either the VIPS mutant passed the test (pink) or it did not (blue hatch).

To allow fair comparison, the histograms in Figure 7 are normalised so that if a VIPS mutation is executed and causes a change of state at different depths (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 7 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 allocated equal areas. Similarly, a mutation which is executed three times, once at depth 6 and twice at depth 40, will contribute one third to x=6 and two thirds to x=40. Disrupting executions of the same type (pass/failed) at the same depth are stacked on top of each other. For example in Figure 7, 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[11].

The same VIPS data are presented in Figure 8, 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) 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 8): two of them disrupt only at depth 24 (both infect 35 968 times); the other three disrupt at two or three depths but cause disruption 96, 96 and 109 824 times at depth 24, giving in total $35\,968 + 35\,968 + 96 + 96 + 109\,824 = 181\,952$.

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 8) can occur at a range of nesting levels, they predominate at depths greater than about 30. E.g. in Figures 7 and 8, seven independent silent mutations (pink) contribute to the area for depth > 31, compared to one impactful mutation (blue), $p = 3\%$[12]. Figure 9 again shows this impact v. depth data but this time the distribution (minimum, quartiles, mean and maximum) for each individual mutation is gathered together.

[11] Of the 25 randomly sampled VIPS failing mutations, eight introduce a disruption at depth 8. Four of these also cause disruptions at another depth. In this example, these four each disrupt at depth 8 exactly half the time, so giving at x=8, $y = 6 = (4 + 4 \times \frac{1}{2})$ plotted with blue hatching in Figure 7.

[12] $p = \%3$ non-parametric one sided statistical hypothesis sign test.

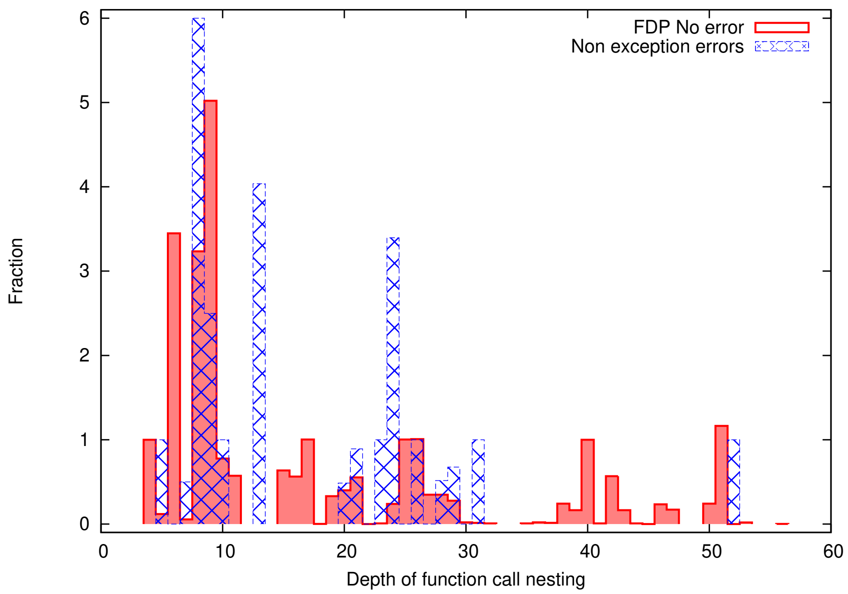

Fig. 7 25 mutations which change internal state but output is unaffected (shaded pink) and 25 which change output (pattern) without raising an exception or reporting an error. (See page 15.)

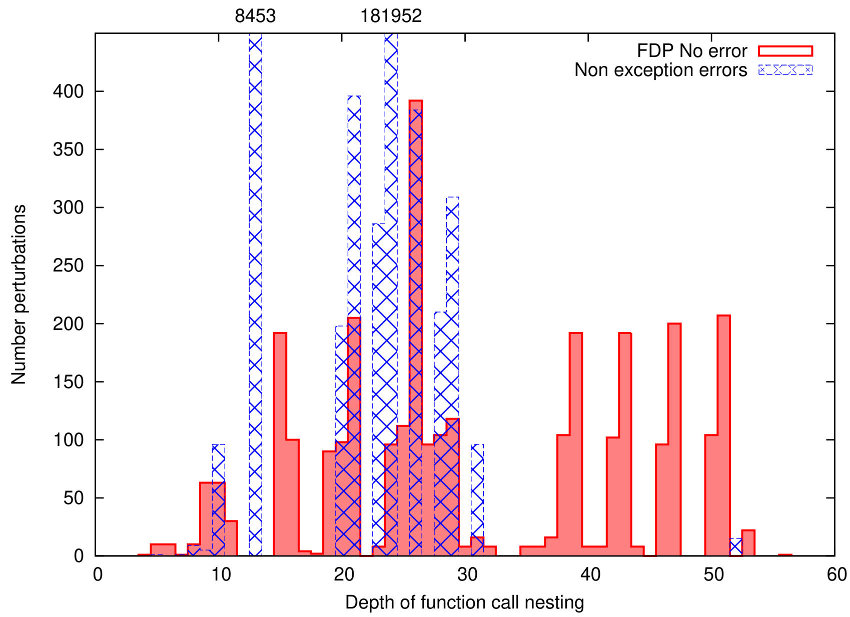

Fig. 8 25 mutations with no impact and 25 which change output. Same data as Figure 7. The vertical axis is truncated to 0–450, as otherwise perturbations which cause errors to the output (blue hatching) nested 13 functions deep x=13 (8453) and x=24 (181952), would dominate all the other data. (Graph described on page 15.)

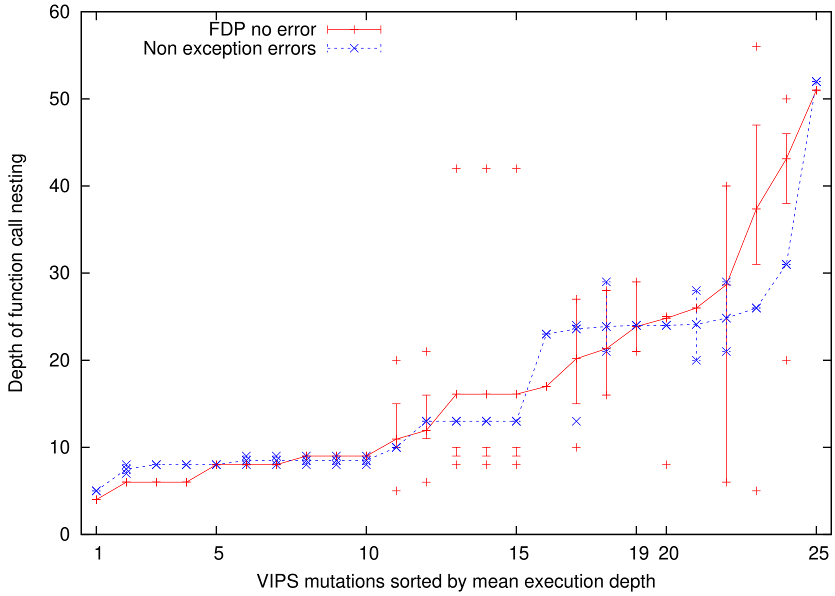

Fig. 9 25 VIPS mutations with no impact (mean depth +) and 25 which change output (mean depth ×). Error bars show interquartile range. +× also show min and max depth. Notice mutations with average depth $y > 30$ tend not to impact VIPS thumbnail output. Same data as Figures 7 and 8.

Table 5 Left: 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. (± indicates standard error). Right: Data for 43 randomly selected gem5 mutants without error. (All instrumented gem5 mutants are known to have either executed or not, i.e. no "na" in row 2.)

VIPS				gem5			
Executed	Infected	count	fraction	Executed	Infected	count	fraction
N	N	45	49% ± 5%	N	N	10	23% ± 7%
na	N	13	14% ± 4%				
y	N	8	9% ± 3%	y	N	8	19% ± 6%
y	y	25	27% ± 5%	y	y	25	58% ± 8%
		91				43	

We can estimate the fraction of Failed Disruption Propagation (FDP) using data gathered from the non-error VIPS mutants when we sought our random sample of 25 mutants which did cause disruption but did not cause an error (see page 14). The left hand side of Table 5 considers 91 uniformly random chosen non-identical mutants of the 438 which run without error (first row Table 2 page 12). 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 $12\% \pm 2\%$ of Magpie mutants show failed disruption propagation.

6.4 gem5 Failed Disruption Propagation (FDP)

To investigate the variation of mutation impact with runtime nesting depth for gem5, we follow the same sampling philosophy for gem5 as we did for VIPS (previous section). That is, we again exclude the 77 (47+17+10+3) mutations which failed at runtime with an exception or where gem5 itself reported an error (13), leaving 55 (Table 4 page 14) where an error was detected only because one or more of the files generated by gem5 were different from those generated by the unmutated code. As with VIPS, we do not distinguish levels of error severity, only that the files are different. We choose uniformly at random 25 of these 55 gem5 mutations and instrument them (blue cross hatching in Figures 10 and 11). Similarly we instrument mutations chosen uniformly at random from the 142 mutations which ran without error. We continue drawing at random until we have 25 mutations which are executed at least once and which change the state or flow of control. Again we use GNU libc `backtrace` to measure the depth of function nesting (plotted with pink in Figures 10 and 11, see also Figure 12 and right hand side of Table 5).

Like VIPS, the instrumented gem5 mutants can be executed many thousands of times (hence use of log scale in Figure 11) and at different depths in the function calling hierarchy. Unlike VIPS, gem5 mutants appear deterministic.

Like Figure 7, the histograms in Figure 10 are normalised so that if a mutation is executed and causes a change of state at different depths the vertical height is plotted in proportion to the number of disrupting executions for that depth (page 15). This ensures that the area allocated to each of the (25+25) mutations plotted in Figure 10 is the same. Thus two mutations which pass all the tests but one is executed half a million times (depth 23) and the other only six times (depth 4), are plotted with the same area. Disrupting executions of the same type (pass/failed) at the same depth are stacked on top of each other. For example in Figure 10, the blue peak (y=7.999) at depth x=30 represents all the failing disruptions at depth 30 across the eight of 25 mutations randomly sampled from the 55 which failed without raising an error or exception.

The same data are presented in Figure 11. Now the vertical (y) axis represents the number of perturbations (on a log scale). That is, the y-axes shows the sum of all the disruptions of the same class (pass/fail) at the same depth. Taking the example of the eight failing mutations (plotted with blue in Figure 11) which change state at depth 30, five only execute at depth 30 and the other three are predominately at depth 30 (total 723 676). The other three also execute at depth 26 but contribute only 222 executions of the 119 470 executions at that depth.

The spike in non-error mutations at depth 31 suggests (as with VIPS) that run time mutations at greater than depth ≈ 30 are less likely to influence the program's output. (In Figures 10 and 11, eleven independent silent mutations (pink) contribute to the area for depth > 30, compared to two impactful mu-

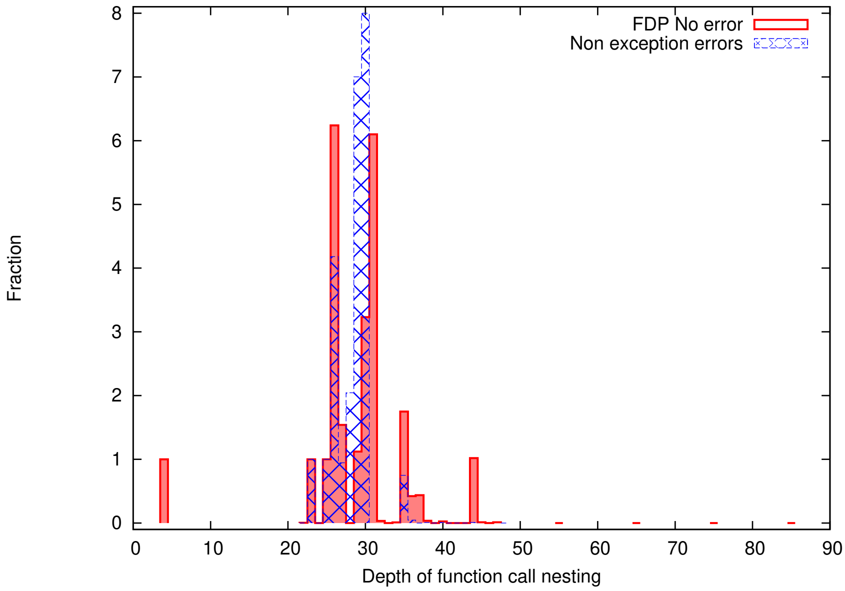

Fig. 10 25 gem5 mutations which change internal state but output is unaffected (shaded pink) and 25 which change output (pattern) without raising an exception or reporting an error. (See page 18.)

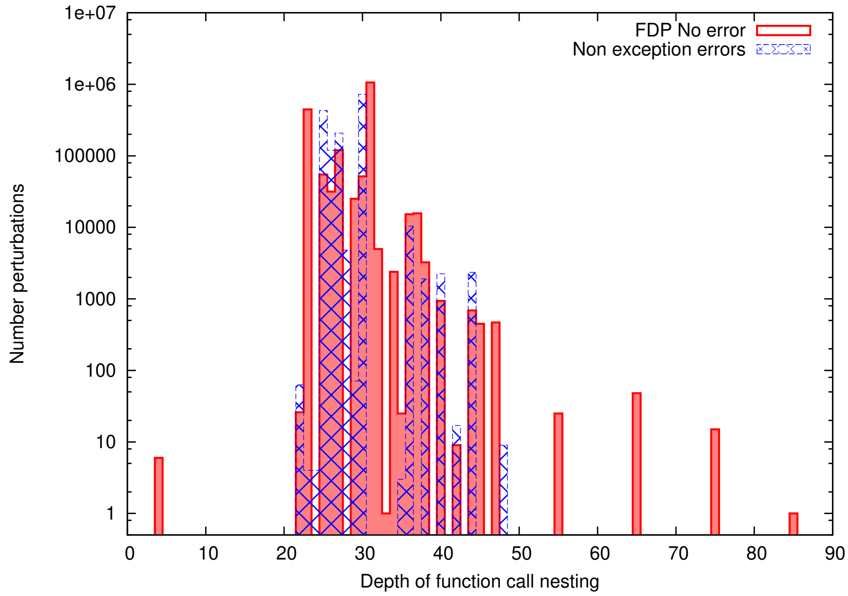

Fig. 11 25 gem5 mutations with no impact and 25 which change output. Same data as Figure 10. Note log vertical scale (Graph described on page 18.)

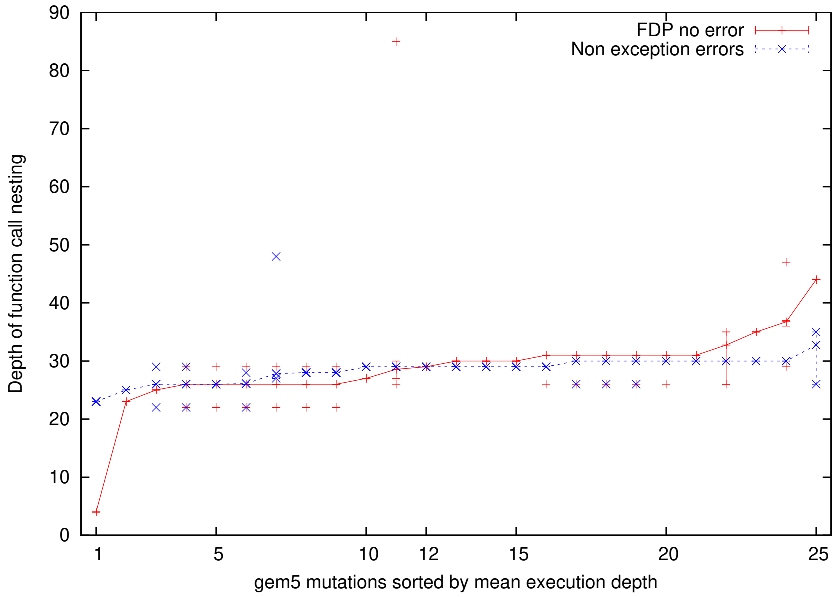

Fig. 12 25 gem5 mutations with no impact (mean depth +) and 25 which change output (mean depth ×). Error bars show interquartile range. +× also show min and max depth. Notice mutations with average depth $y > 30$ tend not to impact any of gem5's outputs. Same data as Figures 10 and 11.

tations (blue), $p = 1\%$[13].) Figure 12 again shows the depth data but this time the distribution (minimum, quartiles, mean and maximum) for each individual mutation is gathered together. If we look at Figure 12 we can see in the deeper half of the mutations there is a (albeit small) tendency for deeper mutations to show failed disruption propagation (FDP). (All 13 equivalent mutants (red) in the top half of Figure 12 have deeper means than the top 13 mutations which impact the output (blue), $p = 0.01\%$[14].)

As with VIPS, we can estimate the fraction of failed disruption propagation from our random sample of 25 gem5 mutants which did cause disruption but did not cause an error (see pages 14 and 15). The right hand side of Table 5 considers 43 uniformly at random chosen non-identical mutants of the 142 which run without error (first row Table 4). 25 of the 43 are executed and disrupt the program but do not change the output. This is 58% of the sample, or 82.6 ± 11 of 142. If we exclude automatically detected equivalent Magpie mutations (last column Table 4), this is 82.6 of 515. That is, $16\% \pm 2\%$ of non-equivalent Magpie gem5 mutants show failed disruption propagation (FDP).

[13] $p = \%1$ non-parametric one sided statistical hypothesis sign test.
[14] $p = 0.01\%$ non-parametric one sided statistical hypothesis sign test.

7 Discussion

We start the discussion with Magpie, the GI tool we use to generate source code changes. We suggest ways to improve Magpie but conclude the fraction of identical patches (Section 7.1) and the number of variables moved out of scope (Section 7.2) are not too expensive. Sections 7.3 and 7.4 suggest our benchmarks are typical of a wide range of software. Of course all cases of failed disruption propagation (FDP) show information loss, nevertheless in Section 7.5 we describe in detail ten examples of FDP, explaining the various mechanisms which prevent disruption impacting the program's output, so making the software robust.

7.1 Magpie identical patches

The second row of Tables 2 and 4 shows 9% of Magpie mutations are identical to the original code. It is therefore no surprise that they compile, run and generate identical output. (In the second example, gem5, we do not bother running them.) Identical mutations are produced by XML operations ArithmeticOperatorSetting, ComparisonOperatorSetting and NumericSetting:

ArithmeticOperatorSetting has only 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. Similarly XML operation ComparisonOperatorSetting has only 6 choices (`<`, `<=`, `!=`, `==`, `>`, `>=`) and NumericSetting can replace 0 with another 0, 1 with 1, or -1 with -1.

We observe 9% (rather than 1/5 etc.) of Magpie mutations not changing the source code as there are several other mutation operators as well as ArithmeticOperatorSetting, ComparisonOperatorSetting and NumericSetting (see Section 5). Although it is possible, the other XML changes are unlikely to replace the original XML with an identical copy.

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 Schulte (2014), Blot et al (2015), Ting Hu et al (2020), so removing them would change population dynamics, however it seems in general that removing them would not have a deleterious effect.

7.2 Undeclared variable compilation errors

We saw in Section 6.1 that 18% of VIPS mutants fail to compile because the mutation has moved an existing variable out of scope. In gem5, Section 6.2, the fraction of compilations attempted which fail due to bad use of variable names is even higher at 31%. However, usually a mutation failing to compile is a relatively cheap part of the fitness function. Nonetheless the fraction of scope errors could potentially be addressed by:

– Restricting XML based mutations to copying source material within the same source file Langdon and Harman (2015).
– Addition of new Magpie scope validity checks Langdon and Harman (2014).
– Use SBSE Harman and Jones (2001) search techniques (such as genetic programming) to fix up variable names Marginean et al (2015).

Moreover, as we did previously, e.g. Langdon and Alexander (2023), 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.

7.3 Are VIPS and gem5 typical?

Typically both VIPS and gem5 are 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. gem5 uses also pointers but like much C++ code it also uses "pass by reference" (`&`). That is, in both examples, 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.

We anticipate our technique will be useful for investigating failed disruption propagation (FDP) in other programs. We found FDP occurring at similar depths, suggesting perhaps that it will occur in other deeply nested programs at about the same depth. In future we hope to develop light weight analysis tools to highlight particular source code regions of rapid entropy loss. In pure functional system we were able to conduct large mutational robustness studies which showed wide individual variability but on average the impact of disruptions fell exponentially with distance, rather than having a sharp cut off. In general we expect large variation in software robustness to individual errors and perturbations but nonetheless we expect as they get more remote from the program's outputs, they will have on average less impact and the program will tend to be more robust to them.

7.4 Few continuous types

Of the 1247 variables declared in our 37 VIPS C files (Section 3.3), only 33 (2.6%) are continuous (float or double) or pointers to continuous variables. While gem5 makes heavy use of its own types and also uses the C++ `auto` keyword, nonetheless it appears in the 25 C++ gem5 files (Section 3.5), only twenty (1.7%) of the 1188 variables declared are continuous (double). In both VIPS and gem5 the other variables are discrete types (e.g. int, char, string and application specific discrete types). Indeed for VIPS 69% and gem5 23% of variables are pointers to discrete variables. Like VIPS and gem5, 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 Langdon (2022c), Langdon and Banzhaf (2022), Langdon (2022e), Langdon (2022d) showed almost complete failure for sizable disruptions to propagate to the output in very deep programs. Note we were concerned with functions evolved by genetic programming Koza (1992), Poli et al (2008), whereas here we deal with real programs written in traditional (imperative) languages (C/C++), with data flows which do not slavishly follow the nested hierarchy of the procedure calls.

7.5 Explanations of ten examples of Failed Disruption Propagation (FDP)

The following examples are of mutations which have no impact despite being **E**xecuted and changing the program's state. I.e., they causes an **I**nfection but it fails to **P**ropagate to the outputs. The following sections (7.5.1–7.5.10) show the impact of randomly chosen VIPS and gem5 FDP mutations (five of each) and explain why their disruption fails to propagate.

Although there are a few similarities, each of the following cases of failed disruption propagation (FDP) is unique. Sometimes disruption is passed via the run time calling hierarchy to other functions (7.5.1, 7.5.2, 7.5.4, 7.5.7), often disruption is past to other parts of the code via shared variables (7.5.3, 7.5.5, 7.5.7, 7.5.10) and sometimes the disruption does not leave the function which has been mutated (7.5.6, 7.5.8, 7.5.9). In some cases information is progressively lost during irreversible operations such as arithmetic, logical expressions and rounding (7.5.4). And in others it is lost suddenly, e.g. by overwriting mutated results (7.5.1 7.5.2) or by multiply by zero (7.5.6) or variables being explicitly deleted (7.5.3) or implicitly deleted when they go out of scope (7.5.4, 7.5.8, 7.5.9) or simply not used (7.5.5, 7.5.7, 7.5.8, 7.5.9). The final example shows a single mutation causing a huge change affecting more than a quarter of a million variables, each comprising both data and pointers and although it impacts runtime, the disruption if bounded by logical expressions and does not leak out into the program's functionality (7.5.10). The common theme in these FDP examples is in real code information loss is due to irreversible actions.

7.5.1 Example VIPS FDP caused by later over write

Figure 13 shows line 270 of `window.c` being mutated so that field `height` of `struct im_window_t` is not set to zero. Instead `height` retains its existing value. The mutation is executed 15 times. The original content of `window->height` is not deterministic, but is not zero between 0 and 12 times. Typically

```
StmtDeletion(('window.c.xml', 'stmt', 78))
                                 -  window->height = 0;
```

Fig. 13 The Magpie mutation operator `StmtDeletion` removes statement 78 from `window.c.xml`, so deleting the initialisation of `window->height` on line 270 in `window.c`. It is executed 15 times at depth 51 and changes state in up to 12 of these 15 times.

the mutation changes state about half the times it is executed. Four lines after the mutation `im_window_new()` calls `im_window_set()` and passes the mutated variable `window` to it. Although `im_window_set()` uses parts of `window` it does not use the value in `window->height`. Instead near its end `im_window_set()` unconditionally overwrites the whole of `window->height`, thus removing all the impact of the mutation.

Notice the original function `im_window_new()` follows good practice in ensuring all seven fields within `struct window` are initialised (including `height`), even though the immediately following code recalculates five of seven of them. Also the mutation is repeatedly executed in non-deterministic code. It often changes the state of a variable, that disruption propagates into a second function but it fails to propagate beyond the second function.

7.5.2 Example VIPS FDP caused by later over write

```
StmtDeletion(('init.c.xml', 'stmt', 38))
                                 -  im->client2 = NULL;
```

Fig. 14 The Magpie mutation operator `StmtDeletion` removes statement 38 from `init.c.xml`, so deleting the initialisation of `im->client2` on line 150 in `init.c`. It is executed 18 times at depths 8 to 42 and changes state in two of these (depths 8 and 9).

Figure 14 gives a mutation similar to the one in the previous section. It shows line 150 of `init.c` being mutated so that field `client2` of `IMAGE struct im` is not set to `NULL` (0). Instead `client2` retains its existing value (typically 0 or `0xffffffff`). Unfortunately it is not possible to be definitive about exactly why leaving `client2` as `0xffffffff` can never have any impact, but we can see in the multi-threaded code where, as in the previous section, `im` is initialised *en-block*, the value `client2` is not used (for example it is passed as argument `dummy` to function `im_start_one()` and as `dummy2` of `im_stop_one()`, neither of which use it) and in `im_generate()` where `client2` is over written.

So again the mutation is executed. In a proportion of cases, the mutation causes a change of state. This is propagated via global variables to code in distant functions but there the disruption is either ignored or lost when the global variable is overwritten.

7.5.3 Example VIPS FDP caused by bounded use

Figure 15 shows line 225 of `im_init_world.c.xml` being mutated so that field `flags` of `struct option_entries[2]` is set to 1 rather than 0 when the program is initialised (i.e. before `main()` is called). `option_entries` is only used in function `im_get_option_group()`, which is called via `g_option_context_`

```
NumericSetting(('im_init_world.c.xml', 'number', 8), '1')
    -  { "vips-tile-height", 'h', 0, G_OPTION_ARG_INT, &im__tile_height,
    +  { "vips-tile-height", 'h', 1, G_OPTION_ARG_INT, &im__tile_height,
```

Fig. 15 The Magpie mutation operator `NumericSetting` changes the ninth (Magpie indexes start at 0) number in `im_init_world.c.xml` from 0 to 1. So changing the initialisation of `static GOptionEntry option_entries[2]` on line 225 in `im_init_world.c`. The mutation is executed once when the program is initialised changing element 2 of `struct` array `GOptionEntry option_entries int` field `flags` from 0 to 1.

`add_group()` from `main()`. `option_entries` is passed to `g_option_group_add_entries()` which incorporates it into its other argument `option_group`. `option_group` is returned to `main()` as pointer `context` (of type `GOptionContext*`). Note the data pointed to by `context` may be disrupted by the mutation. `main()` passes `context` to `g_option_context_parse()`. However the mutation does not change how `g_option_context_parse()` updates its outputs. Then `main()` deletes `context` using `g_option_context_free()`.

Thus the mutation infects state immediately (even before the program starts), that disruption can be transferred between GTK library calls but is not used outside them and is removed by `g_option_context_free()` before `main()` calls `thumbnail()` to generate output.

7.5.4 Example VIPS FDP caused by logic, rounding and scope limits

```
ArithmeticOperatorSetting(('transform.c.xml', 'operator_arith', 21), '+')
                        -  *oy = trn->ic * mx + trn->id * my;
                        +  *oy = trn->ic * mx + trn->id + my;
```

Fig. 16 The Magpie mutation operator `ArithmeticOperatorSetting` replaces `*` by `+` in `transform.c.xml`'s `operator_arith` 21, so updating the calculation of `*oy` of line 122 in `transform.c`. It is executed and changes state 380 times at depth 26. The mutation is in parallel multi-threaded code and so the order (but not the values) of the mutated calculations varies between runs.

Figure 16 shows the calculation of `double oy` being mutated on line 112 of `transform.c` 380 out of 384 times it is executed. Typically the value of `oy` in function `im__transform_invert_point()` is changed by about 0.5% but there are cases when it is approximately doubled and four cases where it should be 0 but is instead 1.015625.

`im__transform_invert_point()` is passed as a `transform_fn` function pointer to `transform_rect()`, which calls it four times each time it itself is called, once for each corner of a rectangular part of the output thumbnail 128×96 image. In `transform_rect()` the x,y values of each corner calculated by `im__transform_invert_point()` (including the mutated y value) are returned to it as `double`. `transform_rect()` deliberately converts to `int` in order to "Round-to-nearest to try to stop rounding errors growing images." `transform_rect()` combines four `double` x,y point pairs to give a bounding box. Notice taking the maximum or minimum of four numbers loses information as only the extreme of the four values contributes to the output. `double top` and `double` `bottom` each combine four mutated y values. Similarly rounding continuous values to integers also loses information.

`double top` is rounded to give output `int top`. Whilst output `int height` is calculated by rounding `bottom - top`. After rounding `int height` is never disrupted by the mutation. Whilst in 33 of 96 bounding boxes `int top` is increased by 1. Notice combining four values and rounding has reduced the disruption by more than ten fold (380 to 33). The mutated rectangles are passed back to `affinei_gen()` via `Rect need`.

In `affinei_gen()` the mutated `need.top` is passed to `im_rect_intersect rect()`, whose output `Rect clipped` is disrupted (in 33 of 96 executions) by the mutation. Although `im_rect_intersectrect()` increases the disruption from just `need` to include `clipped`, only either `clipped.top` or `clipped.height` are disrupted (not both simultaneously) and state (i.e. values in `need` and `clipped`) remains disrupted in 33 executions of `affinei_gen()`. However the disrupted values in `clipped` never cause a change of control flow and like `need` they are deleted at the end of `affinei_gen()` when they go out of scope and so the disruption is contained in `affinei_gen()` (depth 23).

7.5.5 Example VIPS FDP caused by redundant code

```
StmtDeletion(('meta.c.xml', 'stmt', 23))
-  g_value_register_transform_func(
-    type,
-    G_TYPE_STRING,
-    transform_area_g_string );
```

Fig. 17 The Magpie mutation operator `StmtDeletion` removes statement 23 from `meta.c.xml`, so deleting the call of function `g_value_register_transform_func()` from lines 340–343 in `meta.c`. It is executed once at depth 6.

Figure 17 shows lines 340–343 of `meta.c` being deleted so that GTK library function `g_value_register_transform_func()` is not called. The mutation causes a change of flow of control and the function `transform_area_g_string()` will not be registered as the GTK transformation function between `static GType type "im_area"` and `G_TYPE_STRING`. However `transform_area_g_string()` is not used. That is, the mutation causes a changes of state hidden inside the GTK library but it never has any impact.

7.5.6 Example gem5 FDP caused by multiply by zero

```
ComparisonOperatorSetting(('MemoryPowerModel.cc.xml', 'operator_comp', 2), '>')
-  int64_t tRefBlocal = (t.REFB == 0) ? (t.RAS + t.RP) : (t.REFB);
+  int64_t tRefBlocal = (t.REFB > 0) ? (t.RAS + t.RP) : (t.REFB);
```

Fig. 18 The Magpie mutation operator `ComparisonOperatorSetting` replaced `operator_comp` number 2 (i.e. the third comparison, Magpie starts numbering at 0) in `MemoryPowerModel.cc.xml` with $>$. I.e. `==` on line 214 is replaced by $>$. Line 214 is executed 2432 times at depths 22 or 26 or 29. Each time the mutated code sets `tRefBlocal` to 0 instead of the correct value 39.

Figure 18 shows line 214 of `MemoryPowerModel.cc` being mutated and that the value it calculates for local variable `tRefBlocal` is changed from 39 to 0. `tRefBlocal` is only used in the immediately following for loop (lines 217 to 250).

The for loop always iterates eight times. Each iteration `tRefBlocal` is used (on line 223) by function `vdd0Domain.calcTivEnergy()` to set `energy.refb_energy_banks[i]` (i$= 0 \cdots 7$). The first argument of `calcTivEnergy()` is the expression `c.numberofrefbBanks[i] * tRefBlocal`. (This is the only place the value in `tRefBlocal` is used.) However all eight values of the vector `c.numberofrefbBanks` are zero. So the original code multiplied 0 by 39 to give 0. And the mutated code multiplies 0 by 0 to also give 0. Note `energy.refb_energy_banks` is not disrupted by the mutation. Thus although the mutated value of `tRefBlocal` is used $2432 \times 8 = 19\,456$ times it has no impact in any of them and all information about the mutation is destroyed when `tRefBlocal` goes out of scope at the end of function `MemoryPowerModel::power_calc()`.

7.5.7 Example gem5 FDP caused by data not used

```
RelativeNumericSetting(('CAHelpers.cc.xml', 'number', 1), ('(', '+1)'))
-  memTimingSpec.DQSCK + 1     + (memArchSpec.burstLength /
+  memTimingSpec.DQSCK + (1+1) + (memArchSpec.burstLength /
```

Fig. 19 The Magpie mutation operator `RelativeNumericSetting` changed `'number', 1` (i.e. the second number) from 1 to (1+1) in `CAHelpers.cc.xml`. `CAHelpers.cc` line 69 is executed and changes state a total of 206 994 times (at depths 26 or 27 or 31). Each time the mutation means `CommandAnalysis::timeToCompletion()` returns 17 rather than 16 (an increase of 6.25%).

Figure 19 shows line 69 of `CAHelpers.cc` being mutated so that `CommandAnalysis::timeToCompletion()` returns 17 instead of 16. `timeToCompletion()` is called several times by `CommandAnalysis::idle_act_update()` in `CommandAnalysis.cc`, were the mutation's impact is typically propagated into its output variable `idlecycles_act`. That is, `idlecycles_act` may be a several percent bigger than it should be. Meaning `energy.idle_energy_act_banks[i]` and `energy.idle_energy_act` (both in `MemoryPowerModel.cc`) are also a several percent bigger than they should be. Both `energy.idle_energy_act_banks[i]` and `energy.idle_energy_act` are only used in `MemoryPowerModel::power_print()` which prints them out. However `power_print()` is never used, and so although the mutation has been executed and it has made a difference and that disruption has propagated some distance through the C++ code via function call returns and shared values, ultimately it has no external impact despite the mutation impacting internal state more than two hundred thousand times.

```
StmtInsertion(('page_table.cc.xml', '_inter_block', 57),
('page_table.cc.xml', 'stmt', 17))
                  +  auto old_it = pTable.find(vaddr);
```

Fig. 20 The Magpie mutation operator `StmtInsertion` adds a copy of statement 17 to `_inter_block 57`. (Both are in `page_table.cc.xml`.) This adds line 116 `auto old_it = pTable.find(vaddr);` to `page_table.cc`. It is executed 11 times at depth 35, calling `hashtable find()` each time.

7.5.8 Example gem5 FDP addition of unused variable

Figure 20 shows a mutation which adds a line which simultaneously declares a variable `old_it` and initialises it by calling `PTable::find()`. In the GNU standard C++ template library `hashtable`'s `find()` is free of side effects (`find() const`) and new line 116 is the only place in `EmulationPageTable::unmap()` where `old_it` is used. Therefore although the mutation both changes flow of control and program internal state, all its impact is deleted as soon as `old_it` goes out of scope at the end of each iteration of the enclosing while loop. It could be that if this mutation was applied elsewhere, e.g. at a different depth, it would have the same lack of effect.

7.5.9 Example gem5 FDP deletion of empty for loop

```
StmtDeletion(('mmu.cc.xml', 'stmt', 19))
                        -  for (auto tlb : unified) {
                        -     tlb->flushAll();
                        -  }
```

Fig. 21 The Magpie mutation operator `StmtDeletion` removes statement 19 from `mmu.cc.xml`, so deleting lines 91–93 from `mmu.cc`. It is executed 4 times at depth 26 or 35.

Figure 21 shows a mutation which removes a complete for loop which iterates through the GNU standard template library set `unified`. Again the std library function `begin()`, which is used to initialise the loop control variable `tlb`, has no side effects. Since `unified` is empty, in the unmutated code the loop terminates immediately. Note, since the mutation removes the loop iteration test of `tlb`, the mutation changes flow of control. In the unmutated code the change of state associated with creating and initialising `tlb` is lost when `tlb` goes out of scope. Thus although the mutation changes both state and flow of control, its impact does not propagate past where the loop used to be.

7.5.10 Example gem5 FDP change of state impacts runtime not functionality

Figure 22 shows a mutation which removes a complete if statement. The mutation is inside `MemCtrl::pruneBurstTick()`'s while loop and is executed 226 118 times. In most cases `curTick() > *current_it`, so causing `DPRINTF()` and `burstTicks.erase()` to be called. `DPRINTF()` is a debug macro which checks to see if its first argument `MemCtrl` is true. Since debug flag `MemCtrl` is

```
StmtDeletion(('StmtDeletion(('mem_ctrl.cc.xml', 'stmt', 228))
    - if (curTick() > *current_it) {
    -     DPRINTF(MemCtrl, "Removing burstTick for %d\n", *current_it);
    -     burstTicks.erase(current_it);
    - }
```

Fig. 22 The Magpie mutation operator `StmtDeletion` removes statement 228 from `mem_ctrl.cc.xml`, so deleting the `if` on lines 667–670 in `mem_ctrl.cc`. It is executed 226 118 times at depth 30.

not set, `DPRINTF()` does nothing. Whereas in the unmutated code, `burstTicks.erase(current_it)` typically causes parts of `std::unordered_multiset burstTicks` to be deleted. The mutation causes both the flow of control to be changed (for example `curTick()` is no longer called) and also state changes. That is, the expired, i.e. older than `curTick()`, elements of multiset `burstTicks` are no longer erased. Eventually `burstTicks` contains 223 785 `Tick`.

`pruneBurstTick()` is called by `MemCtrl::doBurstAccess()`, which a couple of lines later calls `DRAMInterface::doBurstAccess()`, which calls `DRAMInterface::activateBank()`, which calls `MemCtrl::verifySingleCmd()`.

`burstTicks` is used in `MemCtrl::verifySingleCmd` 223 786 times at depth 31 or 32. However even though `burstTicks` contains many more elements, the count for the current `cmd_tick` (cf. `burstTicks.count(burst_tick)`) is little effected and (as with the unmutated code) it never exceeds 8 (the value of `max_cmds_per_burst`). So both the `Tick inserted` into `burstTicks` and the value returned by `MemCtrl::verifySingleCmd()` are unchanged. That is, the mutation changes flow of control locally but not elsewhere and although it changes state globally, this impacts run time and memory usage but does not propagate to any of the outputs. Note:

- With -O3 (and no instrumentation) g++ seems to make a good of optimising the now pointless while loop in `MemCtrl::pruneBurstTick()`. (The mutated code takes 0.96 seconds v. 0.92 seconds for the original.)
- The mutation means `burstTicks` will continue to grow. In fitness testing we do not see a big increase in the memory needed to run gem5. However in much larger simulations a computer running gem5 might eventually notice the memory problem.

8 Conclusions: Software is Robust, Deeper Code is More Robust

Software is robust to many mutations. If we exclude obviously poor mutations (e.g. those that failed to compile, are identical, or lie in code that is not used) approximately half (73% VIPS, 49% gem5, Tables 2 and 4) of source code mutations run ok and give the right answer.

We use Voas' PIE framework to explain software robustness in terms of information theory and entropy loss (Section 2 and the appendix). If the modified code was **E**xecuted, and it changed the program's internal state (it was **I**nfected) but information about that disruption was not **P**ropagated to any output, including the program's exit status, we call this failed disruption propagation (FDP) and the software is robust to the mutation. Software robustness could also include partial cases where disruption does indeed reach the output but the answer is only changed a little and may still be usable (e.g. Figure 5).

For any disruption to have impact, information about it must reach the program's outputs. Every executed operation from the site of the disruption to the outputs can lose information. In a strictly hierarchical system, if information is lost en-route it cannot be recovered later. That is, information loss is cumulative. Meaning the deeper the nesting of functionality, the more chance there is of information loss. When all information is lost, the disruption has no further impact and cannot change the output. In strictly nested systems we do see progressive information loss and very deep systems can be 100% robust even to very disruptive mutations (Section 2).

As faults may be invisible, robust systems are more difficult to test. 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.

In traditional imperative languages information flow can by-pass the function call hierarchy via shared data. Our C/C++ programs extensively use shared data and we do see examples of mutation induced disruption spreading via global variables. Nonetheless we still see a weak relationship between depth and impact, with FDP more likely in deeper mutations, particularly in our examples when nested more than about 30 function calls deep, leading to mutations which do not change the program's output. This makes deeper code more robust.

Acknowledgements

I am grateful for the help of Aymeric Blot, Brendan Gregg for FlameGraph and our anonymous reviewers.

Appendix

Theoretical Models of Information Loss in Large Expressions

We start by expanding the example of information loss from one addition (page 3) to an example with a mixture of three additions and two multiplications (Figure 23) and then describe a model of information loss in large expressions. Figures 25 to 28 show the model's Gaussian approximation gets rapidly more accurate and plot values for up to 10 000 additions. Indeed the Gaussian approximation, $2 + \log_2 \sigma$, is a reasonable approximation of the entropy (in bits) of many probability distributions (with standard deviation σ). Our information loss model could be extended to include subtraction. Finally we consider potential Log Normal extensions to multiplication and division.

Figures 23 and 24 expand the example in Figure 1 (page 4) to a multiple level expression. The first two levels, and their entropies, are the same as Figure 1. 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 24) and so (as in Section 2) each has entropy $\mathrm{H}_o = 2.88$ 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 24). Notice addition has a smoothing effect and 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. Figure 24 calculates entropy and plots the probability distributions when using 8 bit calculations, so the output is in the range 0–255, rather than 0–364. 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 use mathematics to 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 25 and 28). 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 25 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 27 (note log scales)[15].

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 Gaussian distribution $N(m, \sigma^2)$ 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 26). Regardless of the mean m, if the standard deviation σ is large compared to 255 then mapping the nicely curved distribution will lead to an almost uniform distribution across 0–255 (with an entropy

[15] 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

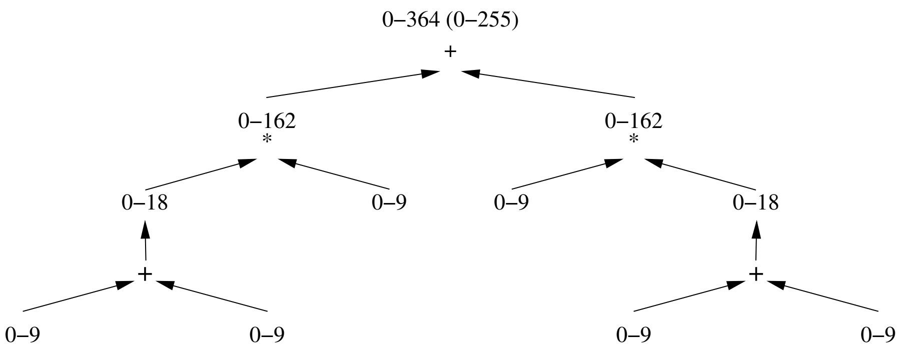

Fig. 23 Intermediate values calculated in expression (a+b)c+d(e+f) where a. . .f are independent random digits (0-9). (If 8 bit precision then output (top) is between 0 and 255.)

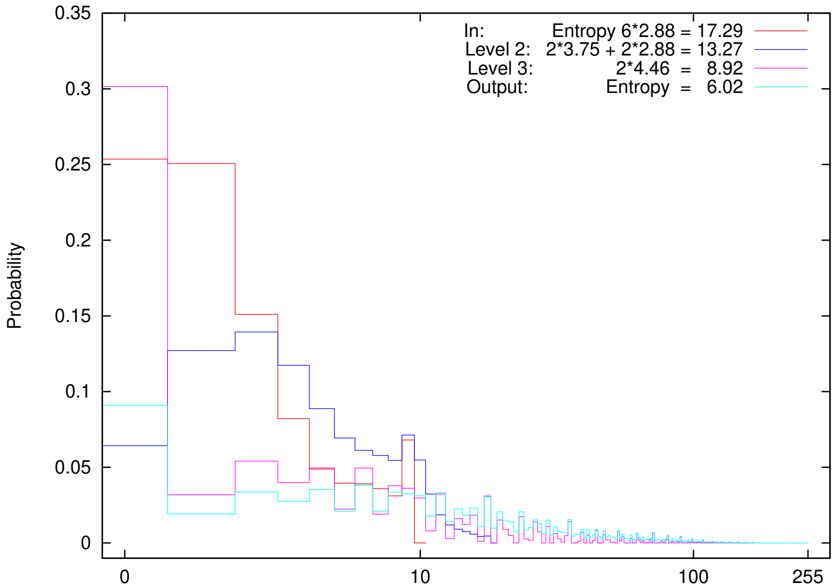

Fig. 24 Distribution of values at each of the four levels in Figure 23. First two plots (red and blue) are the same as the two plots in Figure 1. Note calculations are in 8 bit precision, hence the all values, horizontal axis, line in the range 0–255 (0xFF). Note non-linear x-axis.

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 ≈ 8 bits and the information loss will be $\approx n\mathrm{H}_o - 8$ bits, i.e. almost all the input information is lost. (Again in our example we actually get close to maximum entropy with $\sigma \geq 145$.) The numbers on the right vertical axis of Figure 28 show 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

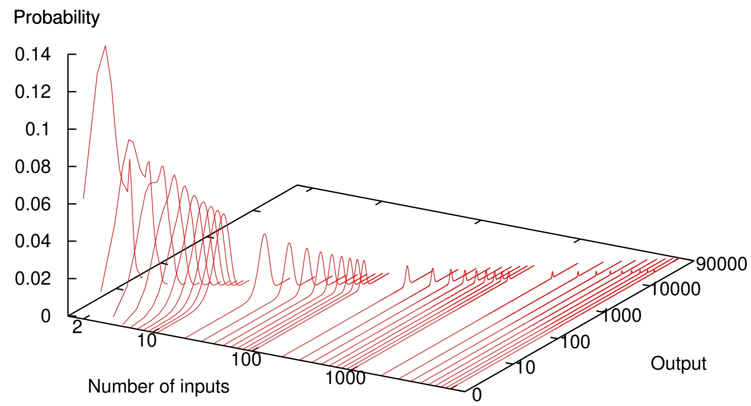

Fig. 25 Distribution of values of adding random digits (0-9) from the VIPS source code. Note non-linear axis.

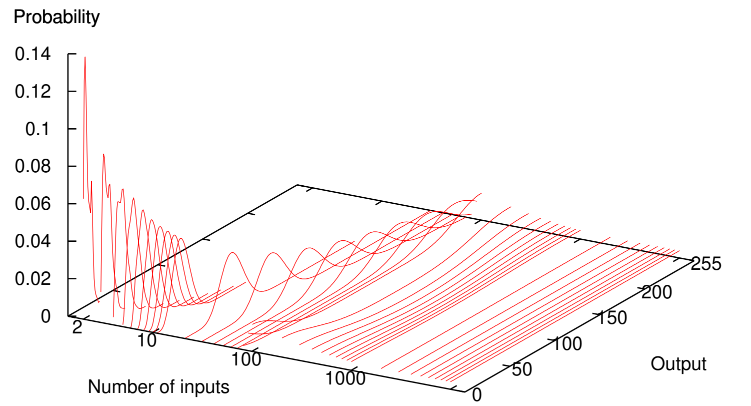

Fig. 26 As Figure 25 but in 8 bits, hence cut off at 255

will approach a Normal distribution, i.e. the product approaches a Log Normal distribution, with known information content as measured by entropy.

Where the log of values are well behaved, like subtraction, we can also extend our argument to division and indeed to expressions composed of both multiplication and division, by simply treating log of division as addition of a negative log value ($\log(a/b) = \log(a) - \log(b) = \log(a) + (-\log(b))$). Again leading to a Normal distribution of the logs and so the output of a product/division expression of independent (logable) terms approaches a Log Normal distribution as it gets bigger. If there as many divisions as multiplies, (i.e. as many $+\log$ as $-\log$) then the mean of the log of the distribution will be zero. That is the distribution will be centered at 1.0 and its entropy will grow $O(\log_2(n))$. That is, much slower than the input entropy $O(n)$.

By keeping track of signs separately, i.e. working with $\mathrm{sign}(a)$ and $\log|a|$ and using $\mathrm{sign}(a \times b) = \mathrm{sign}(\frac{a}{b}) = (\mathrm{sign}(a) + \mathrm{sign}(b))\,\%2$ as well as $\log(|a \times b|) = \log(|a|) + \log(|b|)$ and $\log(|\frac{a}{b}|) = \log(|a|) - \log(|b|)$, we could even extend this argument to negative values. If there are an even number of neg-

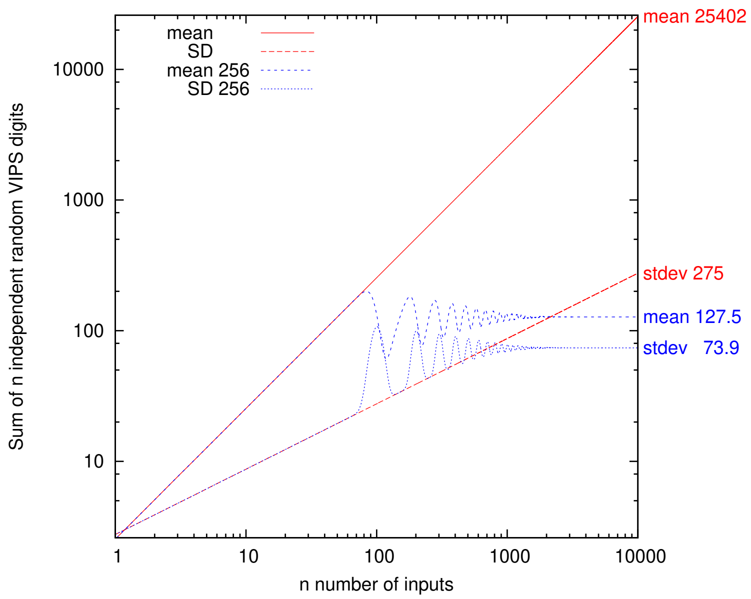

Fig. 27 Mean and standard deviation of adding VIPS 0 to 9 digits. Solid red line mean and red dashed line standard deviation σ assuming unlimited precision when adding n inputs (x-axis) randomly drawn drawn from the distribution of VIPS digits, same data as Figure 25. Note diagonal line shows mean is proportional to n. Whilst $\sigma \propto \sqrt{n}$. Dash blue lines show mean and σ where sum is forced into 8 bit arithmetic and converges to uniform 0–255, entropy 8 bits, same data as Figure 26. Note log-log plot.

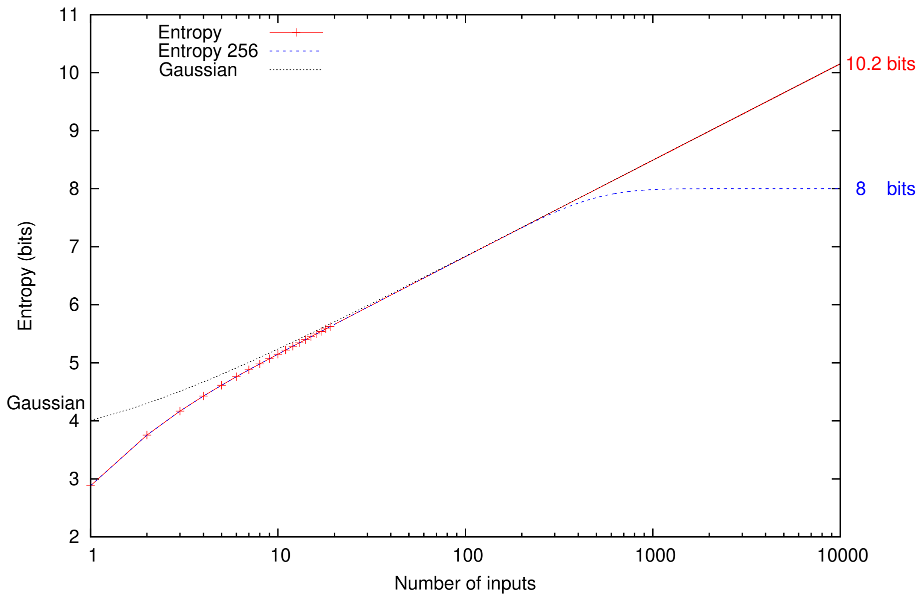

Fig. 28 Entropy of adding VIPS 0 to 9 digits. Solid red + line unlimited precision, same data as Figure 25. Note convergence to Gaussian (dotted line) entropy $\log_2(\sigma) + 2.0471$. (To reduce clutter data above x=20 plotted without crosses +.) Dashed blue line where sum is forced into 8 bit arithmetic and converges to uniform 0–255, entropy 8 bits. Same data as Figure 26, Note log plot.

ative signs, the distribution will be log normal. If odd, it will be -log normal. If signs are distributed evenly at random the distribution will be curiously bimodal: 50% log normal and 50% -log normal. However a continuous distribution that spans both positive and negative values is liable to include zero or at least very small values where finite arithmetic leads eventually to zero.

If randomly drawn input values include non-logable values such as zero they will quickly dominate the distribution of output values. Multiplication by zero gives zero, so even a single zero as input will give zero as output, meaning as the expression gets bigger the probability of it giving a non-zero value shrinks towards nothing. Similarly large expressions with division and zero input will be quickly dominated by how division by zero is treated. Similarly other non-logable values, such as nan (not-a-number) and inf (infinity) will quickly dominate large floating point expressions. With their output distributions depending upon how these exceptional values are treated.

Notice 0 (and nan, inf, etc.) rapidly eat all the information in the other inputs to multiplicative expressions. Depending on exactly how these exception values are handled, as we consider larger expressions, their output probability distribution geometrically converges on a single output value, e.g. 0, which has no information about the inputs and entropy of 0 bits.

References

Androutsopoulos K, Clark D, Haitao Dan, et al (2014) An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: Briand L, van der Hoek A (eds) 36^{th} International Conference on Software Engineering (ICSE 2014). ACM, Hyderabad, India, pp 573–583, URL http://dx.doi.org/10.1145/2568225.2568314

Arcaini P, Yue T, Fredericks EM (eds) (2023) Search-Based Software Engineering - 15th International Symposium, SSBSE 2023, Proceedings, Lecture Notes in Computer Science, vol 14415, Springer, San Francisco, USA, URL http://dx.doi.org/10.1007/978-3-031-48796-5

Bienia C, Kumar S, Singh JP, et al (2008) The PARSEC benchmark suite: characterization and architectural implications. In: Moshovos A, Tarditi D, Olukotun K (eds) 17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008. ACM, Toronto, Ontario, Canada, pp 72–81, URL http://dx.doi.org/10.1145/1454115.1454128

Binkert NL, et al (2011) The gem5 simulator. ACM SIGARCH Computer Architecture News 39(2):1–7. URL http://dx.doi.org/10.1145/2024716.2024718

Blot A, Petke J (2020) Comparing genetic programming approaches for non-functional genetic improvement case study: Improvement of MiniSAT's running time. In: Ting Hu, Lourenco N, Medvet E (eds) EuroGP 2020: Proceedings of the 23rd European Conference on Genetic Programming, LNCS, vol 12101. Springer Verlag, Seville, Spain, pp 68–83, URL http://dx.doi.org/10.1007/978-3-030-44094-7_5

Blot A, Petke J (2021) Empirical comparison of search heuristics for genetic improvement of software. IEEE Transactions on Evolutionary Computation 25(5):1001–1011. URL http://dx.doi.org/10.1109/TEVC.2021.3070271

Blot A, Petke J (2022a) A comprehensive survey of benchmarks for automated improvement of software's non-functional properties. arXiv, URL https://arxiv.org/abs/2212.08540, 2212.08540

Blot A, Petke J (2022b) MAGPIE: Machine automated general performance improvement via evolution of software. arXiv, URL http://dx.doi.org/10.48550/arxiv.2208.02811

Blot A, Aguirre HE, Dhaenens C, et al (2015) Neutral but a winner! how neutrality helps multiobjective local search algorithms. In: Gaspar-Cunha A, Antunes CH, Coello CAC (eds) Evolutionary Multi-Criterion Optimization - 8th International Conference, EMO 2015, Part I, Lecture Notes in Computer Science, vol 9018. Springer, Guimaraes, Portugal, pp 34–47, URL http://dx.doi.org/10.1007/978-3-319-15934-8_3

Bruce BR, Akram A, Nguyen H, et al (2021) Enabling reproducible and agile full-system simulation. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS. , Stony Brook, NY, USA, pp 183–193, URL http://dx.doi.org/10.1109/ISPASS51385.2021.00035

Chen J, Venkataramani G (2016) enDebug: A hardware-software framework for automated energy debugging. Journal of Parallel and Distributed Computing 96:121–133. URL http://dx.doi.org/10.1016/j.jpdc.2016.05.005

Cilibrasi RL, Vitanyi PMB (2007) The google similarity distance. IEEE Transactions on Knowledge and Data Engineering 19(3):370–383. URL http://dx.doi.org/10.1109/TKDE.2007.48

Clark D, Hierons RM (2012) Squeeziness: An information theoretic measure for avoiding fault masking. Information Processing Letters 112(8–9):335–340. URL http://dx.doi.org/10.1016/J.IPL.2012.01.004

Clark D, Langdon WB, Petke J (2020) Software robustness: A survey, a theory, and some prospects. Presented at Facebook Testing and Verification Symposium 2020, URL https://research.facebook.com/blog/2020/11/registration-now-open-for-the-2020-testing-and-verification-symposium/

Dakhama A, Even-Mendoza K, Langdon WB, et al (2023) SearchGEM5: Towards reliable gem5 with search based software testing and large language models. In: Arcaini P, Tao Yue, Fredericks E (eds) SSBSE 2023: Challenge Track, LNCS, vol 14415. Springer, San Francisco, USA, pp 60–166, URL http://dx.doi.org/10.1007/978-3-031-48796-5_14, winner best challenge track paper

DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: Help for the practical programmer. IEEE Computer 11:31–41. URL http://dx.doi.org/10.1109/C-M.1978.218136

Dorn J, Lacomis J, Weimer W, et al (2019) Automatically exploring tradeoffs between software output fidelity and energy costs. IEEE Transactions on Software Engineering 45(3):219–236. URL http://dx.doi.org/10.1109/TSE.2017.2775634

Espinel VA (2016) The $1 trillion economic impact of software. Tech. rep., BSA, The Software Alliance, Washington, DC, USA, URL https://softwareimpact.bsa.org/pdf/Economic_Impact_of_Software_Report.pdf

Gabin An, Jinhan Kim, Shin Yoo (2018) Comparing line and AST granularity level for program repair using PyGGI. In: Petke J, Stolee K, Langdon WB, et al (eds) GI-2018, ICSE workshops proceedings. ACM, Gothenburg, Sweden, pp 19–26, URL http://dx.doi.org/10.1145/3194810.3194814

Gabin An, Blot A, Petke J, et al (2019) PyGGI 2.0: Language independent genetic improvement framework. In: Apel S, Russo A (eds) Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering ESEC/FSE 2019). ACM, Tallinn, Estonia, pp 1100–1104, URL http://dx.doi.org/10.1145/3338906.3341184

Gelperin D, Hetzel B (1988) The growth of software testing. Communications of the ACM 31(6):687–695. URL http://dx.doi.org/10.1145/62959.62965

Haraldsson SO, Woodward JR, Brownlee AEI, et al (2017) Exploring fitness and edit distance of mutated python programs. In: Castelli M, McDermott J, Sekanina L (eds) EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming, LNCS, vol 10196. Springer Verlag, Amsterdam, pp 19–34, URL http://dx.doi.org/10.1007/978-3-319-55696-3_2

Harman M, Jones BF (2001) Search based software engineering. Information and Software Technology 43(14):833–839. URL http://dx.doi.org/10.1016/S0950-5849(01)00189-6

Hynninen T, Kasurinen J, Knutas A, et al (2018) Software testing: Survey of the industry practices. In: Skala K, et al (eds) 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO. IEEE, Opatija, Croatia, pp 1449–1454, URL http://dx.doi.org/10.23919/MIPRO.2018.8400261

Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5):649–678. https://doi.org/doi:10.1109/TSE.2010.62

Koza JR (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA, URL https://mitpress.mit.edu/9780262527910/genetic-programming/

Langdon WB (2003) The distribution of reversible functions is Normal. In: Riolo RL, Worzel B (eds) Genetic Programming Theory and Practice. Kluwer, chap 11, p 173–187, URL http://dx.doi.org/10.1007/978-1-4419-8983-3_11

Langdon WB (2020) Genetic improvement of genetic programming. In: Brownlee AS, Haraldsson SO, Petke J, et al (eds) GI @ CEC 2020 Special Session, IEEE Computational Intelligence Society. IEEE Press, internet, p paper id24061, URL http://dx.doi.org/10.1109/CEC48606.2020.9185771

Langdon WB (2022a) Dissipative arithmetic. Complex Systems 31(3):287–309. URL http://dx.doi.org/10.25088/ComplexSystems.31.3.287

Langdon WB (2022b) Failed disruption propagation in integer genetic programming. In: Trautmann H, et al (eds) Proceedings of the Genetic and Evolutionary Computation Conference Companion. Association for Computing Machinery, Boston, USA, GECCO '22, pp 574–577, URL http://dx.doi.org/10.1145/3520304.3528878

Langdon WB (2022c) Genetic programming convergence. Genetic Programming and Evolvable Machines 23(1):71–104. URL http://dx.doi.org/10.1007/s10710-021-09405-9

Langdon WB (2022d) Open to evolve embodied intelligence. In: Iida F, Hughes J, Abdulali A, et al (eds) Proceedings of 2022 International Conference on Embodied Intelligence, EI-2022, IOP Conference Series: Materials Science and Engineering, vol 1292. IOP Publishing, Internet, Cambridge, p 012021, URL http://dx.doi.org/10.1088/1757-899X/1292/1/012021

Langdon WB (2022e) A trillion genetic programming instructions per second. ArXiv, URL https://arxiv.org/abs/2205.03251

Langdon WB (2023) The end is not clear. Communications of the ACM 66(7):9. URL http://dx.doi.org/10.1145/3596710

Langdon WB, Alexander BJ (2023) Genetic improvement of OLC and H3 with Magpie. In: Nowack V, Wagner M, An G, et al (eds) 12th International Workshop on Genetic Improvement @ICSE 2023. IEEE, Melbourne, Australia, pp 9–16, URL http://dx.doi.org/10.1109/GI59320.2023.00011

Langdon WB, Banzhaf W (2022) Long-term evolution experiment with genetic programming. Artificial Life 28(2):173–204. URL http://dx.doi.org/10.1162/artl_a_00360, invited submission to Artificial Life Journal special issue of the ALIFE'19 conference

Langdon WB, Clark D (2024a) Deep mutations have little impact. In: Gabin An, Blot A, Nowack V, et al (eds) 13th International Workshop on Genetic Improvement @ICSE 2024. ACM, Lisbon, pp 1–8, URL http://dx.doi.org/10.1145/3643692.3648259, best paper

Langdon WB, Clark D (2024b) Genetic improvement of last level cache. In: Giacobini M, Bing Xue, Manzoni L (eds) EuroGP 2024: Proceedings of the 27th European Conference on Genetic Programming, LNCS, vol 14631. Springer Verlag, Aberystwyth, pp 209–226, URL http://dx.doi.org/10.1007/978-3-031-56957-9_13

Langdon WB, Harman M (2014) Genetically improved CUDA C++ software. In: Nicolau M, Krawiec K, Heywood MI, et al (eds) 17th European Conference on Genetic Programming, LNCS, vol 8599. Springer, Granada, Spain, pp 87–99, URL http://dx.doi.org/10.1007/978-3-662-44303-3_8

Langdon WB, Harman M (2015) Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation 19(1):118–135. URL http://dx.doi.org/10.1109/TEVC.2013.2281544

Langdon WB, Harman M (2016) Fitness landscape of the Triangle program. In: Veerapen N, Ochoa G (eds) PPSN-2016 Workshop on Landscape-Aware Heuristic Search, Edinburgh, URL http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/rn1605.pdf, also available as UCL RN/16/05

Langdon WB, Lorenz R (2017) Improving SSE parallel code with grow and graft genetic programming. In: Petke J, White DR, Langdon WB, et al (eds) GI-2017. ACM, Berlin,

pp 1537–1538, URL http://dx.doi.org/10.1145/3067695.3082524
Langdon WB, Lorenz R (2019) Evolving AVX512 parallel C code using GP. In: Sekanina L, Ting Hu, Lourenco N (eds) EuroGP 2019: Proceedings of the 22nd European Conference on Genetic Programming, LNCS, vol 11451. Springer Verlag, Leipzig, Germany, pp 245–261, URL http://dx.doi.org/10.1007/978-3-030-16670-0_16
Langdon WB, Petke J (2015) Software is not fragile. In: Parrend P, Bourgine P, Collet P (eds) Complex Systems Digital Campus E-conference, CS-DC'15. Springer, Proceedings in Complexity, pp 203–211, URL http://dx.doi.org/10.1007/978-3-319-45901-1_24, invited talk
Langdon WB, Brian Yee Hong Lam, Petke J, et al (2015) Improving CUDA DNA analysis software with genetic programming. In: Silva S, et al (eds) GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. ACM, Madrid, pp 1063–1070, URL http://dx.doi.org/10.1145/2739480.2754652
Langdon WB, White DR, Harman M, et al (2016) API-constrained genetic improvement. In: Sarro F, Kalyanmoy Deb (eds) Proceedings of the 8th International Symposium on Search Based Software Engineering, SSBSE 2016, LNCS, vol 9962. Springer, Raleigh, North Carolina, USA, pp 224–230, URL http://dx.doi.org/10.1007/978-3-319-47106-8_16
Langdon WB, Shin Yoo, Harman M (2017a) Inferring automatic test oracles. In: Galeotti JP, Petke J (eds) Search-Based Software Testing, Buenos Aires, Argentina, pp 5–6, URL http://dx.doi.org/10.1109/SBST.2017.1
Langdon WB, Veerapen N, Ochoa G (2017b) Visualising the search landscape of the Triangle program. In: Castelli M, McDermott J, Sekanina L (eds) EuroGP 2017, LNCS, vol 10196. Springer, Amsterdam, pp 96–113, URL http://dx.doi.org/10.1007/978-3-319-55696-3_7
Langdon WB, Petke J, Lorenz R (2018) Evolving better RNAfold structure prediction. In: Castelli M, Sekanina L, Mengjie Zhang (eds) EuroGP 2018: Proceedings of the 21st European Conference on Genetic Programming, LNCS, vol 10781. Springer Verlag, Parma, Italy, pp 220–236, URL http://dx.doi.org/10.1007/978-3-319-77553-1_14
Langdon WB, Petke J, Clark D (2021) Information loss leads to robustness. IEEE Software Blog, URL http://blog.ieeesoftware.org/2021/09/information-loss-leads-to-robustness-w.html
Langdon WB, Al-Subaihin A, Blot A, et al (2023) Genetic improvement of LLVM intermediate representation. In: Pappa G, Giacobini M, Vasicek Z (eds) EuroGP 2023: Proceedings of the 26th European Conference on Genetic Programming, LNCS, vol 13986. Springer Verlag, Brno, Czech Republic, pp 244–259, URL http://dx.doi.org/10.1007/978-3-031-29573-7_16
Lorenz R, Bernhart SH, Höner zu Siederdissen C, et al (2011) ViennaRNA package 2.0. Algorithms for Molecular Biology 6(1). URL http://dx.doi.org/10.1186/1748-7188-6-26
Malan KM (2021) A survey of advances in landscape analysis for optimisation. Algorithms 14:40. URL http://dx.doi.org/10.3390/a14020040
Marginean A, Barr ET, Harman M, et al (2015) Automated transplantation of call graph and layout features into Kate. In: Labiche Y, Barros M (eds) SSBSE, LNCS, vol 9275. Springer, Bergamo, Italy, pp 262–268, URL http://dx.doi.org/10.1007/978-3-319-22183-0_21, winner Gold HUMIE
Martinez K, Cupitt J (2005) VIPS - a highly tuned image processing software architecture. In: Proceedings of the 2005 International Conference on Image Processing, ICIP. IEEE, Genoa, Italy, pp 574–577, URL http://dx.doi.org/10.1109/ICIP.2005.1530120
Mesecan I, Blackwell D, Clark D, et al (2021a) HyperGI: Automated detection and repair of information flow leakage. In: Khalajzadeh H, Schneider JG (eds) The 36th IEEE/ACM International Conference on Automated Software Engineering, New Ideas and Emerging Results track, ASE NIER 2021, Melbourne, pp 1358–1362, URL http://dx.doi.org/10.1109/ASE51524.2021.9678758, 2108.12075
Mesecan I, Gerten MC, Lathrop JI, et al (2021b) CRNRepair: Automated program repair of chemical reaction networks. In: Petke J, Bruce BR, Huang Y, et al (eds) GI @ ICSE 2021. IEEE, internet, pp 23–30, URL http://dx.doi.org/10.1109/GI52543.2021.00014, best paper

Niedermayr R, Wagner S (2019) Is the stack distance between test case and method correlated with test effectiveness? In: Shaukat Ali, Garousi V (eds) Proceedings of the Evaluation and Assessment on Software Engineering, EASE. ACM, Copenhagen, Denmark, pp 189–198, URL http://dx.doi.org/10.1145/3319008.3319021

Papadakis M, Yue Jia, Harman M, et al (2015) Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique. In: 37th International Conference on Software Engineering (ICSE 2015), Florence, pp 936–946, URL http://dx.doi.org/10.1109/ICSE.2015.103

Peng WW, Wallace DR (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, Gaithersburg, MD 20899, USA, URL https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-209.pdf, CODEN: NSPUE2

Petke J, Alexander B, Barr ET, et al (2019) A survey of genetic improvement search spaces. In: Alexander B, Haraldsson SO, Wagner M, et al (eds) 7th edition of GI @ GECCO 2019. ACM, Prague, Czech Republic, pp 1715–1721, URL http://dx.doi.org/10.1145/3319619.3326870

Petke J, Clark D, Langdon WB (2021) Software robustness: A survey, a theory, and some prospects. In: Avgeriou P, Dongmei Zhang (eds) ESEC/FSE 2021, Ideas, Visions and Reflections. ACM, Athens, Greece, pp 1475–1478, URL http://dx.doi.org/10.1145/3468264.3473133

Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, URL http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)

Schulte E (2014) Neutral networks of real-world programs and their application to automated software evolution. PhD thesis, University of New Mexico, Albuquerque, USA, URL https://digitalrepository.unm.edu/cs_etds/49/

Schulte E, Dorn J, Harding S, et al (2014) Post-compiler software optimization for reducing energy. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14. ACM, Salt Lake City, Utah, USA, pp 639–652, URL http://dx.doi.org/10.1145/2541940.2541980

Smigielska M, Blot A, Petke J (2021) Uniform edit selection for genetic improvement: Empirical analysis of mutation operator efficacy. In: Petke J, Bruce BR, Huang Y, et al (eds) GI @ ICSE 2021. IEEE, internet, pp 1–8, URL http://dx.doi.org/10.1109/GI52543.2021.00009

Terragni V, Jahangirova G, Tonella P, et al (2020) Evolutionary improvement of assertion oracles. In: Cohen M, Zimmermann T (eds) Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, Sacramento, California, USA, pp 1178–1189, URL http://dx.doi.org/10.1145/3368089.3409758

Ting Hu, Tomassini M, Banzhaf W (2020) A network perspective on genotype-phenotype mapping in genetic programming. Genetic Programming and Evolvable Machines 21(3):375–397. URL http://dx.doi.org/10.1007/s10710-020-09379-0, special Issue: Highlights of Genetic Programming 2019 Events

Veerapen N, Ochoa G (2018) Visualising the global structure of search landscapes: genetic improvement as a case study. Genetic Programming and Evolvable Machines 19(3):317–349. URL http://dx.doi.org/10.1007/s10710-018-9328-1, special issue on genetic programming, evolutionary computation and visualization

Veerapen N, Daolio F, Ochoa G (2017) Modelling genetic improvement landscapes with local optima networks. In: Petke J, White DR, Langdon WB, et al (eds) GI-2017. ACM, Berlin, pp 1543–1548, URL http://dx.doi.org/10.1145/3067695.3082518, best presentation prize

Voas JM, Miller KW (1995) Software testability: The new verification. IEEE Software 12(3):17–28. URL http://dx.doi.org/10.1109/52.382180

Xiangjuan Yao, Harman M, Yue Jia (2014) A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Briand L, van der Hoek A, Jalote P (eds) ICSE. ACM, Hyderbad, pp 919–930, URL http://dx.doi.org/10.1145/2568225.2568265