A Journey Among Java Neutral Program Variants

Nicolas Harrand · Simon Allier · Marcelino Rodriguez-Cancio · Martin Monperrus · Benoit Baudry

Received: date / Accepted: date

Abstract Neutral program variants are functionally similar to an original program, yet implement slightly different behaviors. Techniques such as approximate computing or genetic improvement share the intuition that potential for enhancements lies in these acceptable behavioral differences (e.g., enhanced performance or reliability). Yet, the automatic synthesis of neutral program variants, through speculative transformations remains a key challenge.

This work aims at characterizing plastic code regions in Java programs, i.e., the areas that are prone to the synthesis of neutral program variants. Our empirical study relies on automatic variations of 6 real-world Java programs. First, we transform these programs with three state-of-the-art speculative transformations: add, replace and delete statements. We get a pool of 23445 neutral variants, from which we gather the following novel insights:

We would like to thank Amine Benelallam and Paul Bettega for their feedback on this work. This work has been partially supported by the EU Project STAMP ICT-16-10 No.731529 under the H2020 framework program, and by the Wallenberg Autonomous Systems and Software Program.

Nicolas Harrand
KTH, Sweden
E-mail: harrand@kth.se

Simon Allier
DGA, France
E-mail: simon.allier@intradef.gouv.fr

Marcelino Rodriguez-Cancio
Vanderbildt University, USA
E-mail: marcelino.rguez.cancio@gmail.com

Martin Monperrus
KTH, Sweden
E-mail: martin.monperrus@csc.kth.se

Benoit Baudry
KTH, Sweden
E-mail: baudry@kth.se
developers naturally write code that supports fine-grain behavioral changes; statement deletion is a surprisingly effective speculative transformation; high-level design decisions, such as the choice of a data structure, are natural points that can evolve while keeping functionality. Second, we design 3 novel speculative transformations, targeted at specific plastic regions. New experiments reveal that respectively 60%, 58% and 73% of the synthesized variants (175,688 in total) are neutral and exhibit execution traces that are different from the original.

1 Introduction

Neutral program variants are at the core of automatic software enhancement. The intuition is that these variants that are different from the original, yet are similar have the potential for enhanced performance, security or resilience. Approximate computing explores how programs variants can provide different trade-offs between accuracy and resource consumption [19]. Software diversity aims at using these variants to reduce the knowledge that an attacker can take for granted when designing exploits [1]. Genetic improvement [22] automatically searches the space of program variants for improved performance.

Despite their key role, the automatic synthesis of neutral program variants, is still a major challenge. In this work, we focus on two specific challenging task: understand how and where to transform a program to synthesize a neutral variant. The how part refers to the design of speculative transformations that introduce some behavioral variations. The where refers to the parts of a program that can stand behavioral variations, while maintaining the overall functionality similar to the original program. We call these parts of programs the plastic code regions.

Our work aims at characterizing these plastic code regions that shall be targeted by speculative transformations to synthesize neutral program variants. This journey focuses on Java programs and the in-depth analysis of various speculative transformations on 6 large, mature, open source Java projects. We articulate our journey around three main parts.

First, we run state of the art speculative transformations [27,3] that add, delete or replace an AST node. We consider that a transformation synthesizes a neutral variant if the variant compiles and successfully passes the test suite of the original program. This first contribution is a conceptual replication [29] of the work by Schulte and colleagues [27]. This replication addresses two threats to the validity of Schulte’s results: our methodology mitigates internal threats, by using another tool to detect neutral variants, and our experiment mitigates external threats by experimenting with a new set of programs, in a different programming language.

Second, we analyze a set of 23,445 neutral variants. We provide a quantitative analysis of the types of AST nodes and the type of transformation that more likely yield neutral variants. We analyze the interplay between the synthesis of neutral program variants and the specification of the original program...
provided as a set of test cases. Also, we manually analyze dozens of neutral variants to provide a qualitative analysis about the semantics of plastic code regions play in Java programs.

In the third part of our investigation, we design and experiment with three novel, targeted speculative transformations: ADD METHOD INVOCATION, SWAP SUBTYPE and LOOP FLIP. Our experiments with our 6 Java projects demonstrate a significant increase in the rate of neutral variants among the program variants (respectively 60%, 58% and 73%). We consolidate these results by assessing that the neutral variants indeed implement behavior differences: we trace the execution of these variants, and observe that 100% actually exhibit behavior diversity.

In summary, this work contributes novel insights about neutral program variants, as follows:

- A conceptual replication of the work by Schulte and colleagues [27] about the existence of neutral variants, with a new tool, new study subjects and a different programming language
- A large scale quantitative analysis about the types of Java language constructs that are prone to neutral variants synthesis with the state of the art speculative transformations: add, delete and replace AST nodes
- A deep, qualitative analysis of plastic code regions that can be exploited to design efficient speculative transformations
- Three targeted speculative transformations that significantly increase the ratio of neutral variants, compared to the state of the art
- Open tools and datasets to support the reproduction of the experiments, available at: https://github.com/castor-software/journey-paper-replication

The rest of this paper is organized as follows. In section 2, we define the terminology for this work. In section 3, we introduce the experimental protocol that we follow in order to investigate the synthesis of neutral variants. In subsection 4.1, we analyze the types of speculative transformations and AST nodes that more likely yield neutral variants. In subsection 4.3, we manually explore and categorize neutral variants according to the role of the code region that has been transformed. In subsection 4.5, we leverage the analysis of previous sections to design novel speculative transformations targeted at specific code regions. In section 5, we discuss some key findings of this study. Section 6 elaborates on the threats to the validity of this work. Section 7 discusses related works and we conclude in section 8.

2 Background and Definitions

Here we define the key concepts that we leverage to explore the different regions of Java programs that are prone to the synthesis of neutral program variants.
2.1 Generic speculative transformations

![Figure 1](image-url)

**Fig. 1** Generic speculative transformations

Given an initial program, which comes along with a test suite, we consider three generic speculative transformations on source code that have been defined in previous work [27,3,22]. These transformations operate on the abstract syntax tree (AST). First, we randomly select a statement node in the AST, we check if it is covered by one test case at least (to prevent transforming dead code), then, we consider three types of transformations (cf. Figure 1).

**Definition 1 Speculative transformations.** We consider the following three transformations on AST nodes

- delete the node and the subsequent subtree (delete, Figure 1(c));
- add a node just before the selected one (add, Figure 1(b));
- replace the node and the subtree by another one (replace, Figure 1(d)).

**Definition 2 Transplantation point.** The statement at which we perform a speculative transformation is called the transplantation point.

**Definition 3 Transplant.** For add and replace, the statement that is copied and inserted is called the transplant statement.

We add further constraints to the generic speculative transformations in order to increase the chance of synthesizing neutral variants. For add and replace, we consider transplant statements from the same program as the transplantation point (we do not synthesize new code, nor take code from other programs). We also consider the following two additional steps:

- We build the type signature of the transplantation point: the list of all variable types that are used in the transplantation point and the return type of the statement. The transplant shall be randomly selected only among statements that have a compatible signature.
- When injecting the transplant (as a replacement or an addition to the transplant), the variables of the statement are renamed with names of variables of the same types that are in the scope of the transplantation point.
Figure 3 shows an excerpt of program, in which we have selected one transplantation point. Figure 2 is a transplant example, i.e., an existing statement extracted from the same program. In order to insert the transplant at the transplantation point, we need to rename the variables with names that fit the namespace. The expression \( \text{inAvail} < \text{max} \) can be rewritten in 4 different ways: each integer variable can be replaced by one of the two integer variable identifiers (a or i). The statement \( \text{context.eof} = \text{true} \); can be rewritten in one single way, rewriting \( \text{context.eof} \) into \( b \).

```java
if (inAvail < max) {
    context.eof = true;
}
```

Fig. 2 Transplant

```java
class A {
    int i = 0;
    void m(int a) {
        boolean b = false;
        [...]
        if (this.i < a) {
            b = true;
        }
        [...]
    }
    //Transplantation point
    [...]
}
```

Fig. 3 Transplantation point

```java
class A {
    int i = 0;
    void m(int a) {
        boolean b = false;
        [...]
        if (this.i < a) {
            b = true;
        }
        [...]
    }
}
```

Fig. 4 Transformed code

There are different reasons for which a random add or replace fails at producing a compilable variant. Hence we introduce different preconditions to limit the number of meaningless variants.

For REPLACE, we enforce that: a statement cannot be replaced by itself; for both ADD and REPLACE, statements of type case, AST nodes of type variable instantiation, return, throw are only replaced by statements of the same type; the type of returned value in a return statement must be the same for the original and for its replacement.

2.2 Neutral variant

Given a program \( P \) and a test suite \( TS \) for \( P \), a speculative transformation can synthesize a variant program \( \tau(P) \), which falls into one of the following categories: (i) \( \tau(P) \) does not compile; (ii) the variant compiles but does not pass all the tests in \( TS \): \( \exists t \in TS | \text{fail}(t, \tau(P)) \); (iii) the variant compiles and passes the same test suite as the original program: \( \forall t \in TS | \text{pass}(t, \tau(P)) \). This work focuses on the latter category, i.e., all variants that are equivalent to the original modulo the test suite. We call such variants neutral variants.
Definition 4 Neutral variant. Given a program \( P \), a test suite \( TS \) for \( P \) and a program transformation \( \tau \), a variant \( \tau(P) \) is a neutral variant of \( P \) if the two following conditions hold: 1) \( \tau(P) \) results from a speculative transformation on a region of \( P \) that is covered by at least one test case of \( TS \); 2) \( \forall t \in TS \mid \text{pass}(t, \tau(P)) \)

This work aims at characterizing the code regions of Java programs where speculative transformations are the most likely to synthesize neutral variants.

3 Experimental protocol

Speculative transformations are instrumental for automatic software improvement, and code plasticity is the property of software that supports these transformations. In what follows, we design a protocol to analyze the interplay between transformations, the programming language and code plasticity.

3.1 Research Questions

Our journey among neutral variants is organized around the following research questions:

RQ1. To what extent can we generate neutral variants through random speculative transformations?

This first question can be seen as a conceptual replication of Schulte and colleagues’ [27]’s experiment demonstrating software mutational robustness. Here, we analyze the same phenomenon with a new transformation tool, new study subjects and in a different programming language.

RQ2. To what extent does the number of test cases covering a certain region impact its ability to support speculative transformations?

This question addresses the interplay between the synthesis of neutral variants and the specification for specific code regions. Since our notion of neutral variant is modulo-test, we check if the number of test cases that cover the transplantation point influences the ability to synthesize a neutral variant.

RQ3. Are all program regions equally prone to produce neutral variants under speculative transformations?

In this question, we are interested in analyzing whether the type of AST node or the type of transformation has an impact on the success rate. For instance, it may happen that loops are more plastic than assignments. We study three dimensions in the qualification of transformations: 1) how they are applied (addition of new code versus deletion of existing code); 2) where they are applied, i.e. the type of the transplantation points (e.g. conditions versus method invocations); and 3) for ADD and REPLACE, the type of the transplant.
RQ4. What roles the code regions prone to neutral variant synthesis plays in the program?

This question relies on a manual inquiry of dozens of neutral variants from all programs of our dataset, to build a taxonomy of good program neutral variants. Here, we categorize different roles that certain code regions can play (e.g., optimization or data checking code) and relate this role to the plasticity of the region.

RQ5. Can speculative transformations target specific plastic code regions in order to increase their capacity at synthesizing neutral variants that exhibit behavioral variations?

We exploit the insights gained in RQ3 and RQ4 to define novel types of speculative transformations, which refine the ADD and REPLACE generic transformations: ADD METHOD INVOCATION, SWAP SUBTYPE, LOOP FLIP. These transformations perform additional code analysis to select the transplantation point. This question investigates whether this refinement helps to reduce the number of variants that are not neutral program variants and hence cannot be used as candidates for improvement.

3.2 Protocol

In this paper, we perform the following experiment.

The experiment is budget-based: we try neither to exhaustively visit the search space nor to have a fixed-size sample. Since the investigation of neutral variants is an expensive process, our computation platform is Grid5000, a scientific platform for parallel, large-scale computation [5]. We submit one batch for each program that is run as long as resources (CPU and memory) are available on the grid. Then, for each variant that compiles, we extract or compute the metrics described in previous section. We also manually analyze dozens of neutral variants in order to build a taxonomy of plastic code regions.

In the second part of our study, we refine the speculative transformations defined above, in order to target specific code regions. We run another round of experiments to determine the impact of targeted transformations on the success rate.

3.3 Dataset

We consider the 6 programs presented in Table 1. All programs are popular Java libraries developed by either the Apache foundation, Google or Eclipse [1]. The second column gives the number of classes, the third column the number of statements. This latter number approximates the size of the search space for our speculative transformations. Column 4 provides the number of test case executions when running the test suite and column 5 gives the statement

---

1 The exact versions of the library and the whole dataset is available here: [https://github.com/castor-software/journey-paper-replication/tree/master/projects](https://github.com/castor-software/journey-paper-replication/tree/master/projects)
coverage rate. (This number of test case execution corresponds to the number of junit test methods as reported by maven.)

The programs range between 60 and 666 classes. All of them are tested with very large test suites that include hundreds of test cases executing the program in many different situations. One can notice the extremely high number of test cases executed on commons-collection. This results from an extensive usage of inheritance in the test suite, hence many test cases are executed multiple times (e.g., test cases that test methods declared in abstract classes). The test suites cover most of the program (up to 96% statement coverage for commons-codec). Jgit is the exception (only 70% coverage): it includes many classes meant to connect to different remote git servers, which are not covered by the unit test cases (due to the difficulty of stubbing these servers) This dataset provides a solid basis to investigate the role plastic code regions play to produce modulo-test equivalent program variants.

### 3.4 Metrics

**Definition 5 Success Rate (SR)** is the ratio between the number of neutral variants and the number of transformations that produce a variant that compiles: $\frac{\text{NeutralVariants}}{\text{Compile}}$.

The success rate is a key metric to capture the plasticity of a code region: the higher it is for a certain region, the more this region can be used by speculative transformations to synthesize valid variants. From an engineering perspective, it is good to generate as many neutral variants as possible in any given amount of time. To this extent, it is better to maximize the success rate.

We collect the following metrics to characterize the regions where we perform speculative transformations.

**Definition 6 Transplantation point features:** Let us call $tp$ the transplantation point yielding the neutral variant. We focus on the following features: 1) $TC_{tp}$ is the number of test cases that execute $tp$. 2) $\text{Transform}_{tp}$ is a categorical feature that characterizes the type of transformation that we performed on $tp$: ADD, DELETE or REPLACE. This can be further refined by considering the type of AST node where the transformation occurs.
3.5 Tools

To conduct the experiments described in this paper, we have implemented a tool that runs speculative transformations on Java programs and automatically runs a test suite on the variant, in order to select neutral variants. This tool, Sosieifier is open source and available online. The analysis and transformation of the JAVA AST mostly relies on another open source library called Spoon.

To capture, align and compare execution traces described in subsection 4.5, we have implemented yajta, a library to tailor runtime probes and trace representations. It uses a Java agent, which instruments Java bytecode with Javassist, to collect log information about the execution. Scalability is a key challenge here, since the insertion of probes on every branch of every method represents a considerable overhead both in terms of execution time, and heap size. For example, a single test run can generate a trace up to GBs of data, which turns into a performance bottleneck when comparing the traces from hundreds of variants. This is especially true for performance test cases such as PhoneticEnginePerformanceTest (335 500 702 method calls and 990 617 578 branches executed) in commons-codec. These issues are well described in the work of Kim et al.

Consequently, we optimized the tracing process as follows: i) execute and compare only the test cases that actually cover the transplantation point in the original program; ii) add transformation-specific knowledge to target the logs (e.g. the addition of a method invocation only requires to trace method call); and iii) collect and store complete traces only for the original program, and compare this trace with the variant behavior on-the-fly. This way, we determine, at runtime, if a divergence occurs and we do not need to store the execution trace of the variant

4 Results

4.1 Success rate of random transformations

This section focuses on RQ1.

RQ1. To what extent can we generate neutral variants through random transformations?

We run speculative transformations on our six case studies (cf. Table 1). Table 2 gives the key data about the neutral variants computed with the budget-based approach described in subsection 3.2. It sums up the results of the 180 207 variants generated, from which 98225 compile and 23445 are
neutral variants. The second, third and fourth columns indicate the number of neutral variants synthesized by \textsc{add}, \textsc{delete} or \textsc{replace}. The fifth column indicates the global success rate (SR) as defined in Definition 5, i.e., the rate of neutral variants among all variants that we generated and that compile. The last column (exploration) indicates the rate of program statements on which we ran a transformation, i.e., the extent to which we explored the space of transplantation points. The low exploration rate for \textsc{jgit} is related to the large size of the project: since our exploration of speculative transformations has a bounded resource budget, we could not cover a large program as much as a small one.

The data in Table 2 provides clear evidence that it is possible to synthesize neutral variants with speculative transformations. In other words, it is possible to speculatively transform statements of programs and obtain programs that compile and are equivalent to the original, modulo the test suite. The program variants that compile are neutral variants in up to 30\% of the cases (for \textsc{jgit}).

This first research question is a conceptual replication of the study of Schulte and colleagues \cite{27}. Their speculative transformations are the same as ours. Yet, they ran experiments on a very different set of study subjects: 22 programs written in C, of size ranging from 34 to 59K lines of code and with test suites of various coverage ratios (from 100\% to coverage below 1\%). They also run experiments on the assembly counterpart of these programs. Their results show that 33.9\% of the variants of on C code at neutral, with a standard deviation of 10. They also obtain 39.6\% of neutral variants at the assembly level, with a standard deviation of 22 on assembly variants.

Our results confirm the main observation Schulte and colleagues: running speculative \textsc{add}, \textsc{delete} and \textsc{replace} randomly can synthesize a significant ratio of neutral program variants. The success ratio between both our and Schulte’s experiments are of the same order of magnitude. Their experiments generate slightly more neutral variants, which could indicate that different programming languages allow various degrees of plasticity. In particular, a stronger type system can limit code plasticity. Yet, the in-depth analysis of differences between languages is outside the scope of this paper.

Table 2 Success rate for the synthesis of neutral program variants with the generic, random speculative transformations

|                  | add | del | rep | SR  | exploration |
|------------------|-----|-----|-----|-----|-------------|
| commons-codec    | 289 | 146 | 266 | 18.0\% | 91.9\% |
| commons-collections | 3912 | 754 | 3960 | 21.8\% | 83.3\% |
| commons-io       | 1754 | 319 | 1472 | 21.1\% | 92\% |
| commons-lang     | 419 | 190 | 537 | 15.7\% | 78\% |
| gson             | 2190 | 215 | 1897 | 25.3\% | 80.3\% |
| jgit             | 1924 | 1375 | 2963 | 30.0\% | 57\% |
| total            | 10078 | 2809 | 10558 | 23.9\% | - |

The data in Table 2 provides clear evidence that it is possible to synthesize neutral variants with speculative transformations. In other words, it is possible to speculatively transform statements of programs and obtain programs that compile and are equivalent to the original, modulo the test suite. The program variants that compile are neutral variants in up to 30\% of the cases (for \textsc{jgit}).

This first research question is a conceptual replication of the study of Schulte and colleagues \cite{27}. Their speculative transformations are the same as ours. Yet, they ran experiments on a very different set of study subjects: 22 programs written in C, of size ranging from 34 to 59K lines of code and with test suites of various coverage ratios (from 100\% to coverage below 1\%). They also run experiments on the assembly counterpart of these programs. Their results show that 33.9\% of the variants of on C code at neutral, with a standard deviation of 10. They also obtain 39.6\% of neutral variants at the assembly level, with a standard deviation of 22 on assembly variants.

Our results confirm the main observation Schulte and colleagues: running speculative \textsc{add}, \textsc{delete} and \textsc{replace} randomly can synthesize a significant ratio of neutral program variants. The success ratio between both our and Schulte’s experiments are of the same order of magnitude. Their experiments generate slightly more neutral variants, which could indicate that different programming languages allow various degrees of plasticity. In particular, a stronger type system can limit code plasticity. Yet, the in-depth analysis of differences between languages is outside the scope of this paper.
Answer to RQ1: Speculative transformations, applied in random code regions, can synthesize neutral program variants on Java source code. The ratio of neutral variants varies between 15.7% and 30.0%, out of thousands of variants, for our dataset. These new results confirm the main observations of Schulte and colleagues.

4.2 Sensitivity to the test suite

RQ2. To what extent does the number of test cases covering a certain region impact its ability to support speculative transformations?

Here, we check if the number of test cases that cover a statement affects the plasticity that we observe. In other words, we evaluate the importance of the number of test cases that cover a transplantation with respect to the probability of synthesizing a neutral variant when we transform that point with one of our speculative transformations.

In order to analyze this impact, we look at the distribution of success rate for all trials made on statements covered by a given number of test cases. Yet, in all projects, the distribution of statements according to the number of test cases that cover it is extremely skewed: more than half of the statements are covered by only one test case and then there is a long tail of few statements that are covered by tens and even hundreds of test cases.

Figure 5 represents the following information, given any transplantation point at which we synthesized one or multiple variants that compile, what is the probability that we succeed in getting a neutral variant, given the number of test cases that cover the transplantation point? Because of the skewed distribution of statements with respect to the number of covering test cases, we group data in bins of transplantation points that represent at least 4000 transformations. Bins for low numbers of test cases cover a narrower range of values because statements covered by few tests are more common than statements covered by an large amounts of tests.

The broken line represents the average success rate per bin of transplantation points. Boxes represent the first and last quartile and the median for the distribution of success rate for statements covered by $n$ test cases. Circles represents outliers (outside of a 95% confidence interval) statement for each classes. For example, for the 5943 transplantation points covered by 1 test case, the weighted average success rate is 26.9% and 25% of these points support the synthesis of neutral variants in more than 37.5% of the trials. Outliers are transplantation points for which success rate is above 93.8%.

For 17 out of 28 bin, the median success rate is 0%, meaning that, for at least half of the transplantation points, none of the variants are actually neutral variants variant is found. Meanwhile, the first quartile is above 0% for
all bins. This means that we successfully synthesized neutral variants for at least 25% of statements covered, independently of the amount of test cases (for 11 bins it is actually more than 50% of statements). The average success rate is close to the overall success rate of 23.9%, whatever the number of test cases covering the transplantation point.

Under the assumption of a linear model, the part of the success rate explained by the number of test cases is negligible (Adjusted R-squared: 0.002036). This implies either that the ability to synthesize a neutral variant on a given statement is not significantly influenced by the number of test cases that cover it with a linear model.

**Answer to RQ2:** the number of test cases that cover a transplantation point is independent from the ability to synthesize a neutral variant at
this point. We believe that this indicates the presence of inherent code plasticity, a concept for which we propose a first characterization in the RQ5. To some extent, the success rate on transplantation points that are covered by large numbers of test cases reflects this amount of software plasticity. It may even be the very first quantitative measure of it.

4.3 Language level plasticity

RQ3 Are all program regions equally prone to produce neutral variants under speculative transformations?

Table 3 Distribution of Statement type across projects

| Node Type            | Min | Med | Max |
|----------------------|-----|-----|-----|
| Invocation           | 34% | 37% | 39% |
| Assignment           | 17% | 19% | 22% |
| Return               | 10% | 13% | 19% |
| If                   | 9.4%| 10% | 14% |
| ConstructorCall      | 4.2%| 6.6%| 8.8%|
| UnaryOperator         | 3.1%| 3.8%| 8.6%|
| Throw                | 1.7%| 2.9%| 4.4%|
| Case                 | 0.13%| 1.2%| 2.6%|
| For                  | 0.55%| 0.76%| 1.5%|
| ForEach              | 0.37%| 0.72%| 0.87%|
| Try                  | 0.17%| 0.65%| 1.4%|
| While                | 0.40%| 0.62%| 0.85%|
| Break                | 0.18%| 0.54%| 1.6%|
| Continue             | 0.018%| 0.21%| 0.65%|
| Switch               | 0.033%| 0.17%| 0.32%|
| Synchronized         | 0   | 0.048%| 0.21%|
| Enum                 | 0   | 0.042%| 0.094%|
| Do                   | 0   | 0.032%| 0.091%|
| Assert               | 0   | 2.68e-03%| 0.0014%|

As a preliminary step for our analysis of the plasticity of language structures, we analyze the usage frequency of each construct. Table 3 summarizes the usage distribution of each construct listed by decreasing median frequency. It appears that 6 constructs are frequently used, in approximately the same proportion in all projects (the top 6 lines of the table). There is no surprise here; these constructs correspond to the fundamental statements of any object-oriented program (assignment, if, invocation, return, constructor call and unaryOperator).

The 13 other constructs present in the table are an order of magnitude less frequent than the top constructs. They are also used in more various
ways across programs. For instance, commons-collections favors `for-each` and `while` loops, while commons-codec uses `for` loops. This can be explained by the different types of structure that these projects use: collections vs arrays. The use of `switch` and its child nodes (break, case, and continue) as well as `try` are also unequally distributed across projects. This disparity partly explains the variation in the observations presented in the following section: uncommon constructs lead to more variations.

4.3.1 ADD

![Figure 6](image_url) Success rate for the ADD transformation, depending on the type of AST node used as transplant

Figure 6 displays the success rate of the ADD transformation according to the type of statements added (type of the transplant node in the AST). Each
The first striking observation is that success rates reach significantly high values. In four cases, the random addition of statements yields more than 60% neutral variants: add “if” nodes in jgit, add “loop” nodes in gson and “try” nodes in both commons-io and jgit. The addition of such nodes provides important opportunities to explore alternative executions.

We observe important variations between node types as well as between projects. However, some regularities emerge: for instance, adding a “return” always yields a low success rate. This low plasticity of return statements matches the intuition: this is the end point of a computation and it is usually a region where a very specific behavior is expected (and formalized as an assertion in the test). Meanwhile, the addition of “Try” statements appears as an effective strategy to generate neutral variants.

Looking more closely at Figure 6, we realize that in average, the addition of “assignment” nodes is the most effective (if we exclude addition of “try” nodes for which we don’t have enough data for all project). This can be explained by the fact that there are many places in the code where the variable declaration and the first value assignment for this variable are separated by a few statements. In these situations it is possible to assign any arbitrary value to the variable, which will be canceled by the subsequent assignment. Yao and colleagues observed a similar phenomenon of specific assignments that “squeezes out” a corrupted state [32]). Also, for some projects, such as commons-io and jgit, the addition of “invocation” nodes is effective. It probably indicates a non-negligible proportion of side-effect free methods in the program, but further experimentation on that matter are detailed in subsection 4.5.1. The addition of conditionals and loops is also effective. It is important to understand that a large number of these additional blocks have conditions such that the execution never enters the body of the block, meaning that only the evaluation of the condition is executed.

4.3.2 DELETE

Figure 7 shows the success rate of the DELETE transformation in function of the type of the AST node deleted, grouped by project. The figure only shows the node types for which enough data were collected (More than 25 transformations tried). While we observe large variations between projects for a given node type, we also note that there is a large variation in the success rate per node type. For instance, this figure suggests that method invocations are less specified than while-blocks, since the success rate is higher.

It appears that deleting a method invocation produces above average results for all projects of our sample. We explain this effect by the presence of side-effect free methods which can be safely removed (discussed also in the next section) and by the existence of many redundant calls (discussed in next section).
The deletion of “continue” nodes is quite effective to synthesize neutral variant as it yield 27% success overall (Not included on the graph since not enough trials were conducted per project, even if overall 102 trials were done.). Those nodes are usually used as shortcuts in the computation, hence removing them yields slower yet acceptable program variants; we discuss this in depth in the next section.

4.3.3 REPLACE

A REPLACE transformation can be seen as the combination of a DELETE and an ADD. Consequently, results are somewhat similar to the ones of ADD and DELETE. The success rate can be seen as the probability that the outcome of a transformation that compiles also passes the tests. This means that if ADD and DELETE transformations were independent for a given statement, the success
rate for REPLACE should be close to the product of the two others. Yet, for each project (as shown in Table 4), the success rate for REPLACE is higher than this product, meaning that local success rate of ADD and DELETE are probably not independent.

|               | ADD SR | DELETE SR | REPLACE SR |
|---------------|--------|-----------|------------|
| commons-codec | 45.3%  | 20.0%     | 10.5%      |
| commons-collections | 53.1% | 23.6% | 13.6% |
| commons-io     | 51.7%  | 19.3%     | 12.5%      |
| commons-lang   | 42.5%  | 13.0%     | 11.0%      |
| gson           | 48.0%  | 18.6%     | 16.7%      |
| jgit           | 58.3%  | 26.7%     | 23.9%      |
| total          | 52.3%  | 23.7%     | 15.7%      |

Table 4 Success rate of ADD, DELETE, and REPLACE by project

We note two key phenomena. First, picking a transplant and a transplantation point that are method invocations is quite effective. This suggests the presence of alternative yet equivalent calls. This is similar to what is discussed in the next section and also by Carzaniga et al. [6]. It also appears that replacing an assignment by another one is efficient. Second, we observe a certain plasticity around “return” statements: some of them can be replaced by the statement surrounded by a “try” or a condition. This suggests the existence of similar statements in the neighborhood of the transplantation point, which perform additional checks.

Answer to RQ3: Generic, random speculative transformations can yield more than 23% neutral program variants, but not all code regions are equally prone to neutral variant synthesis. In particular, method invocations and variable assignments are more plastic than the rest of the code.

4.4 Role of plastic code regions

This section focuses on RQ4. Now, we are interested in understanding whether there is a difference in nature between the neutral variants and the variants that fail the test suite.

RQ4. What roles do the code regions prone to neutral variant synthesis play in the program?

For each program, we selected neutral variant among extreme cases: those synthesized on transplantation points covered by a single test case or synthesized on points covered by the highest number of test cases. By doing this, we are able to build a taxonomy of neutral variants.
This analysis is the result of more than two full weeks of work, where we have manually analyzed dozens of neutral variants. At a very coarse grain, before explaining them in detail, we distinguish three kinds of neutral variants: (i) **revealer neutral variants** indicate the presence of software plasticity in the code; (ii) **fooler neutral variants** are named after Cohen’s [10] counter-measures for security. (iii) **buggy neutral variants** are made on transplantation points that are poorly specified by the test suite, the transformation simply introduces a bug.

**Revealer neutral variants** take their denomination from the fact that they reveal something in the code that is implicit otherwise: *code plasticity*. Once those regions are revealed, speculative transformation can target them, with a high confidence that the variant shall be neutral.

**Fooler neutral variants** are called like this in reference to the “garbage insertion” transformation proposed by Cohen [10]. These neutral variants add garbage code that can fool attackers who look for specific instruction sequences. To this extent, neutral variant synthesis can be seen as a realization of Cohen’s transformation.

**Buggy neutral variants** are simply the degenerated and uninteresting by-products resulting from of weak test cases. We will not provide a taxonomy of buggy neutral variants.

In the following, we discuss categories of revealer and fooler neutral variants. For each category, we present a single archetypal example from the ones synthesized for this work (Table 2). Each example illustrates the difference in the original that produces a neutral variant. Examples come with a table that provides the values for the transplantation point features. A more complete set of examples is available on line\footnote{https://github.com/castor-software/journey-paper-replication/tree/master/RQ4}.

**Plastic specification.** Some program regions implement behavior which correctness is not binary. In other terms, there is no one single possible correct value, but rather several ones. We call such specification “plastic”.

The regions of code implementing plastic specifications provide great opportunities for the synthesis of neutral variants, which transform the programs in many ways while maintaining valuable and correct-enough functionality.

One situation that we have encountered many times relates to the production of hash keys. Methods that produce these keys have a very plastic specification: they must return an integer value that can be used to identify an element. The only contract is that the function must be deterministic. Otherwise, there is no other constraint on the value of the hash key. **Listing 1** illustrates an example of a neutral variant synthesized by removing a statement from a hash method (line 3). To us, the neutral variant still provides a perfectly valid functionality.

**Optimization** Some code is purely about optimization, which is an ideal plastic region. If one removes such code, the output is still exactly the same, only non-functional properties such as performance are impacted. **Listing 2** shows an example of neutral variant that removes an optimization: at the
Listing 1 Delete a statement in `hash` (commons.collection)

```java
int hash(final Object key) {
    int h = key.hashCode();
    - h += (h << 9);
    h ^= h >>> 14;
    h += h << 4;
    h ^= h >>> 10;
    return h;
}
```

end of the if-block (line 7), the original program stores the value of `buf` in `toString`, which allows to bypass the computation of `buf` next time `toString()` is called; the neutral variant removes this part of the code, producing a potential performance degradation if the method is called intensively.

Listing 2 Delete a statement in `toString` (commons.lang)

```java
String toString() {
    String result = toString;
    if (result == null) {
        final StringBuilder buf = new StringBuilder(32);
        [...]
        result = buf.toString();
        - toString = result;
    }
    return result;
}
```

Code redundancy. Sometimes, the very same computation is performed several times in the same program. For instance, two subsequent calls to `list.remove(o)`, even separated by other instructions are equivalent (as long as `list` and `o` do not change between). Speculative transformations naturally exploit this computation redundancy through the removal or replacement of these redundant statements. Replacement with side-effect free also produces valid neutral variants.

Listing 3 displays an example of such a neutral variant (removing if-block at line 3). The statement `if (isEmpty(padStr)) padStr = SPACE;` assigns a value to `padStr`, then this variable is passed to methods `leftPad` and `rightPad`. Yet, each of these two methods include the exact same statement, which will eventually assign a value to `padStr`. So, the statement is redundant and can be removed from the original program, yielding a valid fooler neutral variant. Compared to neutral variants that remove some optimization, those neutral variants might be perform better than the original program.

Implementation redundancy. It often happens that programs embed several different functions that provide the same service, in different ways. For
example, there can exist several versions of the same method with different sets of parameters, which can be used interchangeably by providing good parameter values. It is also possible to use libraries that provide this diversity of similar methods (as demonstrated by Carzaniga and colleagues [6]). Listing 4 illustrates the exploitation of such implementation redundancy inside the program (replace at line 4), i.e., `((Object[]) object)[i]` has the same behavior as `Array.get(object, i)`, with completely different implementations.

**Optional functionality.** In software, not all parts are of equal importance. Some parts represent the core functionality, other parts are about options and are not essential to the computation. Those optional parts are either not specified or the specification is of less importance. These are areas that can be safely removed or replaced while still producing useful variants. Listing 5 is an example of neutral variant that exploits such optional functionality. The neutral variant completely removes the body of the method, which is supposed to transform the type passed as parameter into an equivalent version that is serializable, and instead it returns the parameter. The neutral variant is covered by 624 different test cases, it is executed 6000 times and all executions complete successfully, and all assertions in the test cases are satisfied. This is
Listing 5 Replace in canonicalize (GSon)

```java
public static Type canonicalize(Type type) {
    if (type instanceof Class) {
        Class<?> c = (Class<?>) type;
        return c.isArray() ? new GenericArrayTypeImpl(canonicalize(c.getComponentType())) : c;
    }
    else if (type instanceof ParameterizedType) {
        ParameterizedType p = (ParameterizedType) type;
        return new ParameterizedTypeImpl(p.getOwnerType(), p.getRawType(), p.getActualTypeArguments());
    }
    else if (type instanceof GenericArrayType) {
        GenericArrayType g = (GenericArrayType) type;
        return new GenericArrayTypeImpl(g.getGenericComponentType());
    }
    else if (type instanceof WildcardType) {
        WildcardType w = (WildcardType) type;
        return new WildcardTypeImpl(w.getUpperBounds(), w.getLowerBounds());
    }
    else {
        return type;
    }
}
```

An example of an advanced feature implemented in the core part of GSon that is not necessary to make the library run correctly.

**Fooler neutral variants.** We have realized that a number of ADD and REPLACE transformations result in neutral variants which have more code than the original and where the additional code is harmless for the overall execution. These neutral variants act exactly as Cohen’s “garbage insertion” strategy to fool malicious attackers, hence we call them fooler neutral variants.

We found multiple kinds of fooler neutral variants: some add branches in the code or redundant method calls or redundant sequences of method invocations. Some others reduce the legitimate input space through additional checks on input parameters. Listing 6 is an example of a fooler neutral variant, which adds a recursive call to `ensureCapacity()` (line 12). This could turn the method into an infinite recursion, except that in the additional recursive invocation, the value of the parameter is such that the condition of the first if statement always holds true and the method execution immediately stops. The additional invocation adds a harmless method call in the execution flow.

**Discussion** Let us now consider again the transplantation point features given for each neutral variant. Most neutral variants manually identified as
buggy occur on transplantation points covered by a single test case. In other words, the risk of synthesizing bad neutral variants increases when the number of test cases is low.

More interestingly, we realized that valid revealer and fooler neutral variants can be found both on points intensively tested and on weakly tested points. This confirms the intuition we expressed in the previous section: if a region is intrinsically plastic (has a plastic specification or is optional), the number of test cases barely matters, the only fact that the specification and the corresponding code region is plastic explains the fact that we can easily synthesize neutral variants.

Answer to RQ4: We have provided a first classification of plastic code regions according to the role this region plays in a program. The “revealers” indicate plastic code regions [24]. The “foolers” are useful in a protection setting [10]. Our manual analysis shows the variety of roles that code plays in a program. It uncovers the multitude of opportunities that exist to speculatively modify the execution of programs while maintaining a global, acceptable functionality.

4.5 Targeted transformations

RQ5. Can speculative transformations target specific plastic code regions in order to increase their capacity at synthesizing neutral variants that exhibit behavioral variations?

For this question we design three novel, targeted speculative transformations: ADD METHOD INVOCATION that adds an invocation at the transplantation
point, SWAP SUBTYPE that modifies the type of concrete objects that are passed to variables declared with an abstract type, and LOOP FLIP that reverses the order in which a loop iterates over a sequence of elements. These transformations refine the previous ADD, DELETE, REPLACE to target language constructs that are most likely plastic regions. Our intention is to design transformations that are more likely to produce variants that are syntactically correct, pass the same test suite as the original and exhibit a behavior that is different from the original. We assess the effectiveness of each targeted transformation with respect to:

- success rate, as defined in Definition 5
- behavior difference

We assess behavior difference by comparing the traces produced by the original and the neutral variant when running with the same input. For each targeted transformation, we select the relevant trace features that must be collected, in order to tune yajta (cf. subsection 3.5). Then, the traces are aligned up until the first execution of the transformed region. If the traces diverge between a neutral variant and the original, we consider that the speculative transformation has, indeed, yield an observable behavioral difference. This reveals that i) the transformation was performed on code that is not dead; ii) the compiler optimizations did not mask the effect of the transformations; and iii) two different executions can yield the same result.

4.5.1 ADD METHOD INVOCATION

The ADD METHOD INVOCATION transformation leverages the following observation: Figure 6 indicates that the addition of “invocation” nodes is likely to produce neutral variants. We focus on invocations rather than loops or conditions to reduce the risk of synthesizing variants where the added code is not executed. We also exploit the good results obtained when adding “try” blocks.

The ADD METHOD INVOCATION transformation process.

The transformation starts with the selection of a random transplantation point \( \pi \). Then, it builds the set of methods that are accessible from \( \pi \). A method is considered to be accessible if (i) the method is public, protected and in the same package as the class of \( \pi \), or private and in the same class; (ii) if \( \pi \) belongs to the body of a static method, the method called must be static. (iii) if \( \pi \) does not belong to the body of a static method, the inserted invocation must either refer to a static method, or refer to a method member of the class of an object available in the context. (iv) there exists a set of variables in the local context to fit the method’s parameters. Let us notice that we prevent the method hosting \( \pi \) to be selected, as this would create recursive calls likely to produce an infinite loop.

Once a method \( m \) has been selected, we synthesize a transplant in the form of an invocation AST node to insert at the transplantation point. If the return type of \( m \) is not void, a public field is synthesized in the hosting class and the invocation result is assigned to this field. This additional rule aims at forcing
the usage of the invocation’s result and hence at preventing the compiler from
considering the invocation as dead code and removing it [25]. The transplant
is then wrapped into a “try-catch” block.

A formal definition of the transformation is provided in the replication
repository:

**Illustration of the **ADD METHOD INVOCATION** transformation**

Listing 7 illustrates the addition of an invocation of `conditionC0(String,
int)` before the `return` statement. Since `conditionC0` returns a boolean, a
public field of the same type is added to the `DoubleMetaphone` class to consume
the result of the invocation.

Figure 8 illustrates the juxtaposition of two dynamic call trees: the
tree of the execution of `StringEncoderComparatorTest` on the original
isSilentStart method and the tree when running the same test on the trans-
formed method. Each node on the figure represents a method and each edge
represents a method invocation. The temporal aspect of the execution is rep-
resented in two dimensions: method invocations go from top to bottom, and,
if a method invokes several others, the calls on the left occur before those on
the right. The nodes in grey represent calls the parts of the test execution that
are common to both the original and the transformed program. Nodes in light
green (and connected with dashed lines) represent the parts of the execution
added with the transformation.

**Listing 7** Add method invocation in `DoubleMetaphone.java:882` (commons.codec)

```
 882  + public boolean v6482819 = true;
 883  
 884  private boolean isSilentStart (final String value) {
 885   boolean result = false;
 886   for (final String element : SILENT_START) {
 887     if (value.startsWith(element)) {
 888       result = true;
 889     } break;
 890   }
 891  + try {
 892   + v6482819 = conditionC0(this.VOWELS, this.maxCodeLen);
 893   + } catch (Exception v4663426) {}  
 894   return result;
 895  }
```

[5] https://github.com/castor-software/journey-paper-replication/tree/master/RQ5
Searching the space of the ADD METHOD INVOCATION transformation

The size of the search space can be bound by the product of the number of statements in the targeted program and the number of methods it declares. In practice, we limit ourselves to methods for which we can pass parameters within the context of the transplantation point, which significantly reduces the size of the space. Yet, the space remains huge. Consequently, for experimental purposes, we limit our search to up to 10 different methods per transplantation point. If more than 10 methods can be invoked at the same point, we randomly select 10.

Behavior diversity

To assess the behavioral variations introduced by the addition of a method invocation, we use yajta to trace the number of times each method in the program invokes any other method. This observation produces a $N \times N$ matrix, where $N$ is the number of methods executed when running the test suite. The comparison of the matrix produced on the original program and the one produced on the variant reveals if it is, indeed, possible to observe additional method invocations (i.e., additional behavior) at runtime.
Table 5 Call Matrix of the execution of `StringEncoderComparatorTest` when adding a call to `conditionC0` in `isSilentStart` (in `DoubleMetaphone`)

|                             | isSilentStart(String) | conditionC0(String, int) | contains(String, int, String[]) | charAt(String, int) | isVowel(char) |
|-----------------------------|-----------------------|--------------------------|---------------------------------|--------------------|---------------|
| ...                         | ...                   | ...                      | ...                             | ...                | ...           |
| `isSilentStart(String)`     | ...                   | 0 0 12 0 0 0 0 ...       | ...                             | ...                | ...           |
| `conditionC0(String, int)` | ...                   | 0 0 12 0 0 0 0 12 ...    | 0 0 0 0 0 ...                   | ...                | ...           |
| `doubleMetaphone(String, boolean)` | ... | 12 0 0 0 0 0 0 ... | ...                             | ...                | ...           |

Table 5 shows an excerpt of the trace when running `StringEncoderComparatorTest`. Each line records the number of times a method has invoked the methods mentioned in the column header. The results recorded during the execution of the test on the original program appear in black, while the new calls, occurring as a result of the transformation, appear in green. We observe that the transformed method (`isSilentStart`) is called 12 times by `doubleMetaphone` during the test run, on the original program. The speculative transformation adds an invocation to `conditionC0` in `isSilentStart`. This results in 12 invocations of `conditionC0`, as well as 12 times more invocations to all the methods invoked by `conditionC0`. These can be observed in Figure 8 as 12 subtrees of one node calling 3 other appear in green.

**Empirical results for the ADD METHOD INVOCATION transformation**

Table 6 Success rate of add method invocation

|                      | #TPs | #Trials | #NV  | SR  |
|----------------------|------|---------|------|-----|
| commons-codec        | 1722 | 18960   | 11150| 62% |
| commons-collections  | 7027 | 43540   | 26294| 60% |
| commons-io           | 1608 | 12372   | 7532 | 61% |
| commons-lang         | 4287 | 139780  | 86453| 62% |
| gson                 | 2460 | 34880   | 18215| 52% |
| jgit                 | 12822| 38609   | 22840| 60% |
| Total                | 29926| 287232  | 172484| 60% |

Table 6 displays the results per study object: (#TPs) number of transplantation point for which transformation were attempted, number of times we performed the ADD METHOD INVOCATION transformation and produced a
compilable variant (# Trials); number of transformations that yield a neutral variants (# NV); and the success rate (SR). Overall, 60% of the speculative transformations yield a program variant that compiles and passes the suite, which corresponds to 172484 neutral variants in total.

The first key observation is that method invocations are plastic regions, regardless of the original program. The second observation is that the targeted speculative transformation is significantly more effective than a random invocation addition to synthesize neutral variants: 60% on average instead of the 45% success rate of the ADD transformation presented in Figure 6 when inserting method invocation.

Several factors contribute to this successful synthesis of neutral variants. First, the transformation selects the methods to be added, ensuring that it is possible to get valid parameter values in the context of the transplantation point. This design decision can favor repeating an invocation that already exists in the method that hosts the transplantation point. If the method is idempotent, the trace changes with no side effect. Second, the additional invocation is wrapped into “try” blocks. This may also lead to the compilation of invocations that quickly throw an exception and therefore do not cause any state change. In general, the addition of invocations to idempotent or pure methods can make the insertion benign.

| Table 7 | Success rate of add method invocation depending on Transplant and Transplantation point type |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| Static Tp | 69% | 68% | 68% |
| Non static Tp | 58% | 64% | 58% |
| Total | 59% | 66% | 60% |

In Table 7 we provide the cumulative success rates, with respect to the type of method in which the transplantation point is selected (transplantation point (TP) in static or non-static method) and with respect to the type of transplant (invoke a method that inside the same class as the transplantation point or that is external to that class). In this table, we observe a significant difference between the two types of transplantation points: transplantation points in static methods are more plastic (68%) that in non static ones (58%). We hypothesize that this comes from the fact that in the case of a transplantation point inside a static method, the additional invocation can only be towards a static method. Increased success rate in this case could come from the fact the proportion of pure methods is higher among static methods than among regular methods.

We also observe more successful transformations when the transplant is selected outside the class that hosts the transplantation point (66% instead of 59%). We hypothesize that methods invoked in the same class as the transplantation point are likely to be non-pure methods. The transformation selects invocations to methods for which the context of the transplantation point can
provide values to pass as parameters. This means that most of the methods inside the same class can be invoked, whereas in the case of external methods this tends to select methods with no parameter or methods that have only parameters of primitive data types. We hypothesize that this difference in the selection of candidate methods increases the chance to have more pure methods among external than among internal method invocations.

4.5.2 SWAP SUBTYPE

The results of the REPLACE transformation showed that targeting assignment statements yields more neutral variants than on other types of AST nodes. In this section, we introduce a new transformation that refines REPLACE on “Assignment”, leveraging Java interfaces. A common practice in Java consists in declaring a variable typed with an interface. When a developer adopts this practice, she indicates that any concrete object that implements the interface can be assigned to this variable. The existing diversity of available types sharing an interface can be leveraged to fuel our search for neutral variants.

The SWAP SUBTYPE transformation process

This speculative transformation operates on assignment statements that pass a new concrete object to a variable typed with an interface. The transformation replaces the constructor called in such assignments by one of a class implementing the same interface. In the following experiments we have implemented this transformation for classes and interfaces of Java collections.

Illustration of the SWAP SUBTYPE transformation

Listing 8 shows an example of a SWAP SUBTYPE transformation, while Figure 9 illustrates its impact on the dynamic call tree of one test. Nodes in light teal are method invocations from org.apache.commons.collections which were not present before. (They replace previous calls to the Java standard library).

Listing 8 SwapSubType in Lang.java:130 (commons.codec)

```
130  public static Lang loadFromResource(final String languageRulesResourceName, final Languages languages) {
131    final List<LangRule> rules = new ArrayList<LangRule>();
132    * final List<LangRule> rules = new org.apache.commons.collections4.list.NodeCachingLinkedList<LangRule>();
133    [...]  
134  }
```

Searching the space of SWAP SUBTYPE transformations

The search space here is composed of all statements that assign a new concrete object to a variable which type is a collection (see RQ5 for the actual list). This space is small enough to be explored exhaustively. We target on 16 interfaces, which are implemented by 50 classes (some of which implements several interfaces) regrouped in 3 different libraries (java.util,
Fig. 9 Impact of a modification on the call tree of one execution of PhoneticEngineTest.testEncode()

org.apache.commons.collections and net.sf.trove4j). The complete list of interfaces, and their concrete classes, targeted by this transformation is available in the replication repository. While the choice of a concrete collection might be a long planned decision for performances reasons, we believe that in many cases the choice is made by default.

Behavior diversity
To observe the changes introduced by the swap subtype transformation, we use yajta to trace both the methods defined in the classes of the program that is transformed and all the methods in collection classes that are involved in the transformation (the ones at the transplantation point and the ones in the transplants).

The trace comparison procedure is the same as for the ADD METHOD INVOCATION transformation.

Empirical results for the SWAP SUBTYPE transformation
Table 8 presents the results of the SWAP SUBTYPE transformation on each project of our sample. In total, we synthesized 6031 variants on 339 different transplantation points (i.e. collection assignment to a variable typed as an
Table 8 Success Rate of swap subtype

|        | #Tp | #Trials | #NV | SR  |
|--------|-----|---------|-----|-----|
| commons-codec | 21  | 186     | 164 | 88 %|
| commons-collections | 68  | 738     | 450 | 61 %|
| commons-io | 16  | 183     | 177 | 97 %|
| commons-lang | 41  | 544     | 445 | 82 %|
| gson     | 17  | 266     | 221 | 83 %|
| jgit     | 190 | 2992    | 1403| 47 %|
| Total    | 339 | 6031    | 2777| 58 %|

Out of the 6031 variants that compile correctly, 2777 are neutral variants. This represents a global 58% success rate. We notice that the swap subtype transformation yields more than 80% neutral variants for 4 projects. Yet, for jgit and commons-collections, the success rate falls to 47% and 61% respectively. Overall this represents a geometric means of 74%.

A major reason for the lower success rate on commons-collections is the use of inner classes that implement the Collection interface. This happens to create classes that mix the contract of the Collection interface with the contract of the class inside which the Collection interface implementation is defined. For example, the class MultiValueMap$Values implements the iterator of the Collection interface inside MultiValueMap. Listing 9 shows an instantiation of MultiValueMap$Values that was used as a transplantation point for the swap subtype transformation. The original program assigns a MultiValueMap$Values to valuesView. This means that subsequent calls to MultiValueMap$Values.iterator() return the values that are stored in the field map. Now, since vs is a of type Collection, the swap subtype transformation assumes that it can assign it any object typed with an implementation of Collection, e.g. LinkedList in this example. Yet, because a call to iterator() on an instance of LinkedList only iterate over elements that have been added to the instance, all MultiValueMap$Values.iterator() calls return empty iterators which leads to failing tests. Such situations occurred for 113 variants, 0 of which are neutral.

Listing 9 SwapSubType in MultiValueMap.java:326 (commons.codec)

```java
    @Override
    @SuppressWarnings("unchecked")
    public Collection<Object> values() {
        final Collection<V> vs = valuesView;
        return (Collection<Object>) (vs != null ? vs : (valuesView = new Values()));
    }
```

While the number of candidates to be targeted by this transformation is lower than for other transformations, swap subtype affects all subsequent invocation that target the modified variable. Therefore the speculative trans-
formation impacts the generated variant in a more profound way than other transformations. This effect is well illustrated by Figure 9.

In theory, it is possible to swap any valid subtype of an interface when assigning a concrete object to a variable typed with the interface, and this with no effect on the functionality. This property is a direct consequence of the fact that any requirement on the type of a variable is should be expressed in the interface. In other words, SWAP SUBTYPE should be a sound preserving transformation. Indeed, we observe that there exist at least one neutral variant for 71% of the 339 transplantation points targeted by the SWAP SUBTYPE transformation.

However, in practice we observe that is not always the case, and SWAP SUBTYPE is, indeed, a speculative transformation: only 58% of the transformations actually yield a neutral variant.

Listing 10

```java
List<String> fieldNames = getFieldNames(field);
BoundField previous = null;
- Map<String, BoundField> result = new LinkedHashMap<String, BoundField>();
+ Map<String, BoundField> result = new HashMap<String, BoundField>();

for (int i = 0; i < fieldNames.size(); ++i) {
    String name = fieldNames.get(i);
    if (i != 0) serialize = false; // only serialize the default name
    BoundField boundField = createBoundField(context, field, name,
        TypeToken.get(fieldType), serialize, deserialize);
    BoundField replaced = result.put(name, boundField);
    if (previous == null) previous = replaced;
}
```

Listing 10 illustrates an example where the SWAP SUBTYPE transformation fails at producing a neutral variant. Here, the concrete type in the original program is `LinkedHashMap`. This specific implementation of the `Map` interface keeps the entries in the order of insertion. When the `for` loop iterates through the `fieldNames` list, the `result` map is filled such that the elements in `map` are stored in the same order as the elements in `fieldNames`. Now, when the `swap subtypes` transformation assigns a `HashMap` object to `result` instead of a `LinkedHashMap`, the elements of `result` are ordered with respect to their hash value instead of keeping the order of insertion of `fieldNames`. Consequently, subsequent methods that expect a specific order in `result` fail because of this change.

It is important to notice that when we replace `LinkedHashMap` by `org.apache.commons.collections4.map.LinkedMap` in Listing 10, the corresponding variant is neutral, since the substitute types satisfies the required
invariant: elements are kept in order of insertion. More generally, we can say that this zone is plastic, modulo this type invariant.

4.5.3 LOOP FLIP

Swapping instructions is a state of the art transformation used speculatively by Schulte and colleagues [27] or in a sound way for obfuscation [31]. Here we explore a targeted swap transformation, which reverses the order of iterations in loops.

The LOOP FLIP transformation process

We propose a speculative transformation that reverses the order in which for loops iterate over a set of elements. It targets counted loops, i.e., loops for which we can identify a loop counter variable that is initialized with a specific value and which is increased or decreased at each iteration until it does satisfies a condition. The transformation does not necessarily expect a well-behaved counted loop. The transformation makes the loop run the same iterations as the original loop, but for loop index values in reverse order. To achieve this, we need to identify the initial value, the step, and the last value. Listing 12 shows such an example for a simple case. The loop counter is the variable i, its initial value is 0, the step is +1, so the last value is straightforward to determine (srcArgs.length − 1). In this example, the transformation replaces the original loop with one starting from the last value, with a step of −1 and ending when the loop counter reaches the initial value.

The example of Listing 11 is a non-normalized loop that we still handle with LOOP FLIP. The variable i is still the loop counter. Its first value is 0, and its last value is 28 as 32 is not reachable. For the general case, the last value is the last multiple of step smaller than the difference between the upper bound and the starting value. Yet, as we only transform the code in a static way, this expression is directly inserted in the initialization of the loop counter. More implementation details are given in the replication repository.

Listing 11 Loopflip in MemberUtils.java:115 (commons.lang)

```
115  - for (int i = 0; i < srcArgs.length; i++) {
116  + for (int i = srcArgs.length-1; i >= 0; i--) {
```

Listing 12 Loopflip in UnixCrypt.java:87 (commons.codec)

```
87  - for (int i = 0; i < 32; i += 4) {
88  + for (int i = 32 - (((32 - 0) % 4) == 0 ? 4 : (32 - 0) % 4); i >= 0; i -= 4) {
```

Illustration of the LOOP FLIP transformation

In order to illustrate how test execution may be affected by this transformation, Listing 11 details a transformation, a test that cover the transformed code and its execution trace.

https://github.com/castor-software/journey-paper-replication/tree/master/RQ5
Listing 13  Loopflip in BinaryCodec.java:108,123 (commons.lang)

```java
108 public static byte[] toAsciiBytes(final byte[] raw) {
109     if (isEmpty(raw)) {
110         return EMPTY_BYTE_ARRAY;
111     }
112     final byte[] l_ascii = new byte[(raw.length) << 3];
113     for (int ii = 0, jj = (l_ascii.length) - 1; ii <
114         (raw.length); ii++, jj -= 8) {
115         for (int bits = 0; bits < (BITS.length); ++bits) {
116             if (((raw[ii]) & (BITS[bits])) == 0) {
117                 l_ascii[(jj - bits)] = '0';
118             } else {
119                 l_ascii[(jj - bits)] = '1';
120             }
121         }
122     }
123     return l_ascii;
124 }
```

Listing 13 shows an example where the LOOP FLIP transformation yields a neutral variant of the BinaryCodec.toAsciiBytes() method. This method returns an array of bytes, which is filled inside the loop that we transform. The variant is neutral because the array is filled with values which do not depend on the iteration order of the loop, but only on the value of the loop index. We can think of this as filling the table with a set of numbered operations, where the order in which the operations are performed does not matter. Consequently the l_ascii array contains exactly the same thing, whatever the iteration order of the for loop in line 114.

Listing 14  Call to toAsciiBytes in BinaryCodecTest.java:706,709 (commons.codec)

```java
706     bits = new byte[1];
707     bits[0] = BIT_0; //0b00010111
708     l_encoded = new String(BinaryCodec.toAsciiBytes(bits));
709     assertEquals("00010111", l_encoded);
```

Listing 14 shows an excerpt of the test case that specifies the behavior of BinaryCodec.toAsciiBytes(). It calls the method with the binary value 00010111 as parameter and assesses that the return value is the array of bytes that encodes the String "00010111". Figure 10 and Figure 11 show the execution of both the original and transformed method in that context. Round nodes correspond to method calls, squared ones correspond to branches in the order where they are called (from left to right). The branch highlighted in orange (dashed line) corresponds to the line A (in the same color in Listing 13). The branch highlighted in green (dotted line) corresponds to the line B. We can observe that the execution order is indeed reversed.

Searching the space of LOOP FLIP transformations

In the case of this transformation, the search space is composed of for-loops based on an integer index. Since it is fairly small, we exhaustively explore it.
Behavior diversity

The observation of branch executions would not be enough to systematically detect behavioral differences caused by this transformation for every case. Indeed for a loop whose body is composed of a single branch, branches executed do not depend on the index variable, therefore branch observation would fail to detect differences. Thus the simplest observation method is to insert a probe at the beginning of the transformed loop to trace the value of the loop index.

Empirical results for the loop flip transformation

Table 9 summarizes the results of the loop flip experiments. We observe that this speculative transformation is very effective at synthesizing neutral variants. In total, we synthesized 479 neutral variants out of 656 variants, that compiled, targeting each a different for loop, which corresponds to a global success rate of 73%. This success rate varies from 64% to 92% in commons-collections. This is significantly higher than any of the random speculative transformations analyzed previously.

| Library     | #TPs | #Trials | #NV  | SR  |
|-------------|------|---------|------|-----|
| commons-codec | 42   | 42      | 31   | 73% |
| commons-collections | 61   | 61      | 56   | 92% |
| commons-io    | 35   | 35      | 24   | 69% |
| commons-lang  | 227  | 227     | 146  | 64% |
| gson          | 17   | 17      | 11   | 65% |
| jgit          | 274  | 274     | 211  | 77% |
| Total         | 586  | 586     | 427  | 73% |
The high success rate of LOOP FLIP can be explained by the fact that in many cases this transformation processes loops in which there are no loop-carried dependencies \[12\] (e.g., Listing 13). Meanwhile we can also note that both the number of candidates and the success rate vary widely from one project to another. This can be explained by different usages of loops in different projects. For example, if a project uses \texttt{forEach} loops more often than \texttt{for} loops, then the number of candidates for our transformation decreases. Also, \texttt{for} loops are used for different purposes: in some cases this control structure is used to apply the same computation to \(n\) elements that are independent of each other, whereas in other cases it is used to sequence of computations in which each action depends on the previous one. In the former case, the order of the loop iteration does not matter, while in the latter case, flipping loop order is very likely to modify the global behavior.

Listing 15 Altering an ordered loop in ReflectiveTypeAdapterFactory:140 (gson)

```java
140  List<String> fieldNames = getFieldNames(field);
141  BoundField previous = null;
142  Map<String, BoundField> result = new LinkedHashMap<String, BoundField>();
143  [...]
144  for (int i = 0; i < fieldNames.size(); ++i) {
145    String name = fieldNames.get(i);
146    if (i != 0) serialize = false; // only serialize the default name
147    BoundField boundField = createBoundField(context, field, name,
148    TypeToken.get(fieldType), serialize, deserialize);
149    BoundField replaced = result.put(name, boundField);
150    if (previous == null) previous = replaced;
151  }
```

For example, Listing 15 illustrates a LOOP FLIP transformation that yields a variant that is not neutral. This case is similar to the one discussed on Listing 10 when changing the iteration order, the \texttt{result} map is filled in a different order than in the original case. Consequently, the behavior of the method changes, which does not correspond to the expectation of the callers and eventually fails some test cases.

4.5.4 Discussion

In this section we have leveraged the observations made with generic, random speculative transformations, in order to design three new transformations that target code regions which are very likely plastic. When designing these transformations, we also increased the amount of static analysis performed by the transformation, leveraging the strong type system of Java. Overall, these design decisions aim at focusing the search on spaces of program variants with
high densities of neutral variants. The results confirm these higher densities, with success rates of 60% (ADD METHOD INVOCATION), 58% (SWAP SUBTYPE), 73% (LOOP FLIP) that are significantly higher than the rates with generic, random transformations (23.9% overall).

Beyond the results and observations made with these three transformations, the experiments reported here are very encouraging to explore the ‘grey’ zone that exists between sound, syntax and semantic preserving transformations at one extreme and random, generic highly speculative transformations on the other extreme. We believe that in-depth knowledge about the nature of plastic code regions, combined with static code analysis is essential to design transformations that explore spaces of program variants that are behaviorally diverse, while limiting the amount of resources required to explore these spaces.

Answer to RQ5. Speculative transformations targeted at specific plastic code regions are significantly more effective than random transformations at synthesizing program variants, which exhibit visible behavior diversity and are equivalent modulo test suite. This RQ has explored three targeted speculative transformations that yield 60%, 58%, 73% neutral variants.

5 Discussion

Our journey among the different factors that influence the synthesis of neutral program variants has shed the light on several key findings. We have observed that many neutral variants result from the very specific combinations of one speculative transformation on one specific type of language structure. For example, the DELETE transformation in “invocation” nodes is surprisingly effective at synthesizing neutral variants, while it performs very poorly on “loop”. Similarly, the ADD transformation is very effective with “try” nodes, but is very bad with “return”.

These observations are novel and very interesting to design speculative transformations in future works. Yet, we believe that the most intriguing findings of our work relates to regions of the code that are plastic by nature, and not, by chance, because of one specific transformation.

The functional contract of a code region is what ultimately determines if a variant of that region is neutral or not. Such a contract defines a set of properties about the inputs and outputs of the code region, as well as state invariants for that region. Consequently, a contract can be more or less restrictive on the behaviors that implement the contract. Our empirical inquiry of speculative transformations has revealed that some contracts define loose expectations about the behavior of a code region. In turn, these code regions are more plastic than other parts.

Here are three example of code regions with loose contracts:
the contract of a hash function (e.g., the one of Listing 1) loosely specifies the returned value; it only enforces the result to be a deterministic integer only function of information used in equals. In addition, a weak requirement is that this method should avoid collision. This means any transformation, which side effect is to change the return in a deterministic way, yields a variant that fulfills the contract, even if changing the likelihood of collision impacts performances.

the contract over some data ordering. For example data structures that do not impose an order on their elements, or loops with no loop-carried dependence are code regions that have a loose contract. These regions tolerate many types of transformations that change order, for example LOOP FLIP or SWAP SUBTYPE in case where an ordered collection is replaced by another a non ordered one.

optional functionalities (e.g., optimization code). The elective nature of these code regions make them naturally loosely specified. These functionalities are called by other functions, and the functional contract is defined on these other functions, not on the optional ones. All transformations that remove or modify the optional functionality produce program variant that is very likely to satisfy the contract.

In this work, we have used unit test suites as proxies for functional contracts. As discussed in subsection 4.4, this might lead to false positives (variants considered neutral modulo the test suite, but that happen to be buggy variants). Yet, in many cases, this also allowed us to spot inherently plastic code regions that are prone to several speculative transformations, which can synthesize more neutral variants.

6 Threats to validity

We performed a large scale experiment in a relatively unexplored domain: the characterization of plastic code regions. We now present the threats to validity.

While we aim at analyzing code plasticity, we actually measure the rate of neutral variants produced by specific speculative transformations. This can raise a threat to the construct validity of our study, with respect to two concerns: i) the limitation of plasticity to a given transformation, ii) the confinement of changes only to the source code but not to the behavior. We mitigate the first concern through the manual analysis in our answer to RQ4 that emphasizes the presence of real code plasticity and not only plasticity related to a given transformation. To mitigate the second, we analyzed, in RQ5’s answer, the execution traces proving actual, observable differences in execution.

Our findings might not generalize to all types of applications. Depending on the type of applications and the quality of their test suite, the obtained results could change. To minimize the impact of this threat, we selected open

---

6 Oracle’s documentation (Java 8): https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode
source frameworks and libraries because of their popularity, their longevity and the very high quality of their test suites. In addition, we provided an explicit analysis of the impact of tests on the success rate of transformations in subsection 4.2.

Finally, our large scale experiments rely on a complex tool chain, which integrates code transformation, instrumentation, trace analysis and statistical analysis. We also rely on the Grid5000 grid infrastructure to run millions of transformations. We did extensive testing of our code transformation infrastructure, built on top of the Spoon framework that has been developed, tested and maintained for more than 10 years. However, as for any large scale experimental infrastructure, there are surely bugs in this software. We hope that they only change marginal quantitative results, and not the qualitative essence of our findings. Our infrastructure is publicly available on Github.

7 Related works

Our work is related to the seminal work, which analyze the capacity of software at yielding useful variants under speculative transformations. It is also related to the works that exploit speculative (either random or targeted) to improve software. Here, we discuss the key works in these areas, as well as the novelty of our work.

7.1 Plasticity of software

The work on mutational robustness by Schulte and colleagues is a key inspiration for our own work. These authors explore the ability of software to be transformed under speculative, random copy, deletion and swap of AST nodes. Their experiments on 22 small to medium C programs (30 to 60 K lines of code) show that 30% of the transformations yield variants that are equivalent to the original, modulo the test suite. They call this property of software mutational robustness. More recently, this research group demonstrate that the interaction of several neutral mutations can lead a program to exhibit new positive behavior such as passing an additional test. They call this phenomenon positive epistasis.

Our RQ1 can be considered as a conceptual replication of the work by Schulte and colleague. Our results mitigate two threats to the validity of Schulte’s results: our methodology mitigates internal threats, by using another tool to perform speculative transformations, and our experiment mitigates external threats by transforming Java programs (instead of C). Similarly to Schulte, we conclude “that mutational robustness is an inherent property of software”. Yet, our study also provides completely novel insights about the language constructs and the code areas that support mutational robustness

9 https://github.com/castor-software/journey-paper-replication
A Journey Among Java Neutral Program Variants

39

(we call them plastic code regions) and about the effectiveness of targeted transformations to maximize the synthesis of neutral variants.

Recently, Danglot and colleagues have also explored the capacity of software and absorbing speculative state transformations [11]. They explore correctness attraction: the extent to which programs can still produce correct results under runtime state perturbations. In that study the authors rely on a perfect oracle to assess the correctness of outputs, and they observe that many perturbations do not break the correctness in ten subject programs. Our work also shows that program variants can have different traces and still deliver equivalent results (modulo the test suite). Yet, we rely on different transformations and we analyze in-depth the nature of the code regions that can yield neutral variants.

Our work extends the body of knowledge about forgiving code regions [24]. In particular, we find regions characterized by “plastic specifications”, i.e. regions which are governed by a very open yet strong contract. For instance, the only correctness contract of a hashing function is to be deterministic. On the one hand this is a strong contract. On the other hand, this is very open: many variants of an hashing function are valid, and consequently, many modifications in the code result in valid hashing functions.

Some recent work investigate a specific form of software plasticity, referred to as redundancy [7,14,30]. These works consider that a code fragment is redundant with another fragment, in a specific context, if in that context, both fragments lead a program from a given state to an equivalent one through a different series of intermediate state. This is very close to neutral variants, which have diverse visible behavior and yet satisfy the same properties as assessed by the test suite. The key difference between our works is that we investigate speculative transformations to synthesize neutral variants, i.e. increase redundancy, whereas they analyze redundancy that naturally occurs in software systems.

7.2 Exploiting software plasticity

Genetic improvement [22] is an area of search-based software engineering [15], which consists in automatically and incrementally generating variants of an existing program in order to improve non-functional properties such as resource consumption of execution time. All variants should pass the test suite of the original program. Existing works in this domain rely on random speculative transformations to search for program variants: Schulte and colleagues [26] exploit mutational robustness to reduce energy consumption; Langdon et al [18] add, delete, replace nodes in C, C++, CUDA ASTs to improve performance; Cody-Kenny et al. [9] delete AST nodes, searching for performance improvement; López and colleagues explore program mutations to optimize source code. All these works leverage the existence of code plasticity, and the performance of the search process can be improved with targeted speculative transformations. In particular, our results with the SWAP SUBTYPE transforma-
tion, show that changing library is very effective to generate neutral variants, and this transformation is a key enabler to improve performance [2].

Software diversification [4] is the field concerned about the automatic synthesis of program variants for dependability. Existing works in this area also intensively exploit software plasticity and speculative transformations: Feldt [13] was among the first to use genetic programming to generate multiple versions of a program to have failure diversity; we relied on random transformations to synthesize diverse implementations of Java programs [2]; recent work on composite diversification [31], investigate the opportunity to combine multiple security oriented transformation techniques. These works can benefit from our findings about targeted speculative transformations, which introduce important behavior changes (in particular the swap subtype transformation), while maximizing the chances of preserving the core functionality.

Shacham and colleagues [28] and, more recently, Basios and colleagues [2] investigate source code transformations to replace libraries and data structures, in a similar was as the swap subtype transformation. This corroborates the idea of a certain plasticity around these data structures, and the notion of interface.

8 Conclusion

The existence of neutral program variants and the ability to generate large quantities of such variants are essential foundations for automatic software improvement. Our work contributes to these foundations with novel empirical facts about neutral variants and with actionable transformations to synthesize such variants. Our empirical analysis explores the space of neutral variants of Java programs, focusing on 6 large open source projects, from different domains. We generated 98225 variants that compile for these projects, through speculative transformations, and 23445 were neutral variants, i.e., more than 20% of the variants run correctly and pass the same test suite as the original. A detailed analysis of these neutral variants revealed that some language constructs are more prone plastic than others to the synthesis of neutral variants (for example method invocations) and also that some code regions have specific roles that make them plastic (for example optimization code).

The actionable contribution of our work comes in the form of three novel speculative transformations for Java programs. We have designed these transformations to target specific code regions that appear more prone to neutral variant synthesis. Our experiments show that these transformations perform significantly better than generic ones: 60% (add method invocation), 58% (swap subtype), 73% (loop flip) instead of 23.9%.

One key insight from the series of experiments reported in this work is that some code regions are inherently plastic. These code regions are naturally prone to behavioral variations that preserve the global functionality. These regions include code that has a plastic specification (e.g., hash function); optional functionality (e.g., optimization code) or regions that can be
naturally reordered (e.g., loops with no loop-carried dependence). In our future work, we wish to leverage this insight about the deep nature of large programs to develop techniques that can generate vast amounts of software diversity for obfuscation [16] and moving target defenses [20].

References

1. S. Allier, O. Barais, B. Baudry, J. Bourcier, E. Daubert, F. Fleurey, M. Monperrus, H. Song, and M. Tricoire. Multi-tier diversification in web-based software applications. *IEEE Software*, 32, 2015.
2. M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr. Darwinian data structure selection. In *Proc. of ESEC/FSE*, 2018.
3. B. Baudry, S. Allier, and M. Monperrus. Tailored Source Code Transformations to Synthesize Computationally Diverse Program Variants. In *Proc. of ISSTA*, pages 149–159, 2014.
4. B. Baudry and M. Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Computing Survey*, 48(1):16:1–16:26, 2015.
5. R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, et al. Grid’5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.
6. A. Carzaniga, A. Godi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-checking oracles from intrinsic software redundancy. In *Proc. of ICSE*, ICSE 2014, pages 931–942, May 2014.
7. A. Carzaniga, A. Mattavelli, and M. Pezzè. Measuring software redundancy. In *Proc. of ICSE*, volume 1, pages 156–166, May 2015.
8. S. Chiba. Load-time structural reflection in java. In *Proc. of ECOOP*, pages 313–336. Springer, 2000.
9. B. Cody-Kenny, M. O’Neill, and S. Barrett. Performance localisation. In *Proc. of GI*, 2018.
10. F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.
11. B. Danglot, P. Preux, B. Baudry, and M. Monperrus. Correctness attraction: a study of stability of software behavior under runtime perturbation. *Empirical Software Engineering*, pages 1–34, 2017.
12. P. S. Devan and R. Kamat. A review-loop dependence analysis for parallelizing compiler. *International Journal of Computer Science and Information Technologies*, 5(3):4038–4046, 2014.
13. R. Feldt. Generating diverse software versions with genetic programming: an experimental study. *IEEE Proceedings - Software*, 145(6):228–236, Dec 1998.
14. M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proc. of FSE*, pages 147–156. ACM, 2010.
15. M. Harman and B. F. Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.
16. N. Harrand and B. Baudry. Software diversification as an obfuscation technique. In *International Workshop on Obfuscation: Science, Technology, and Theory*, pages 31–34, 2017.
17. D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu. Dual execution for on the fly fine grained execution comparison. *SIGPLAN Not.*, 50(4):325–338, Mar. 2015.
18. W. B. Langdon and J. Pelaez. Software is not fragile. In *First Complex Systems Digital Campus World E-Conference 2015*, pages 203–211. Springer, 2017.
19. S. Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys*, 48(4):62, 2016.
20. H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding focus in the blur of moving-target techniques. *IEEE Security & Privacy magazine*, 12(2):16–26, Mar 2014.
21. R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46:1155–1179, 2015.

22. J. Petke, S. Haraldson, M. Harman, D. White, J. Woodward, et al. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.

23. J. Renzullo, W. Weimer, M. Moses, and S. Forrest. Neutrality and epistasis in program space. In *Proc. of GI*. ACM, 2018.

24. M. Rinard. Obtaining and reasoning about good enough software. In *Proc. of DAC*, pages 930–935. ACM, 2012.

25. M. Rodriguez-Cancio, B. Combemale, and B. Baudry. Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding. In *Proc. of ASE)*, Singapore, Singapore, Sept. 2016.

26. E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 639–652. ACM, 2014.

27. E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

28. O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. *SIGPLAN Not.*, 44(6):408–418, June 2009.

29. F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo. The role of replications in Empirical Software Engineering. *Empirical Software Engineering*, 13(2):211–218, Apr. 2008.

30. M. Suzuki, A. C. de Paula, E. Guerra, C. V. Lopes, and O. A. L. Lemos. An exploratory study of functional redundancy in code repositories. In *Proc. of SCAM*, pages 31–40. IEEE, 2017.

31. S. Wang, P. Wang, and D. Wu. Composite software diversification. In *Proc. of ICSME*, pages 284–294. IEEE, 2017.

32. X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proc. of ICSE*, pages 919–930, 2014.