Student:
Benjamin DANGLOT

Supervisors:
Martin MONPERRUS
Philippe PREUX
Xavier LEPALLEC

CORRECTNESS ATTRACTION: EXPLORING THE PERTURBABILITY OF SOFTWARE

Wednesday 2nd November, 2016

informatiques mathématiques
CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille
Université de Lille 1 SCIENCES ET TECHNOLOGIES

Contents

Introduction

In statics, a branch of mechanics, one learns in the introductory course that there are two kinds of equilibrium: stable and unstable. Consider Figure 1, where a ball lies respectively in a basin (left) and on the top of a hill (right). The first ball is in a stable equilibrium, one can push it left or right, it will come back to the same equilibrium point. On the contrary, the second one is in an unstable equilibrium, a perturbation as small as an air blow directly results in the ball falling away.

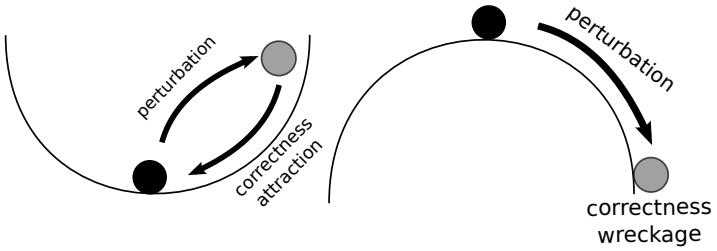

Figure 1: The concept of stable and unstable equilibrium in physics motivate us to introduce the concept of "correctness attraction".

In one of his famous lectures [1], Dijkstra has stated that in software, *"the smallest possible perturbations – i.e. changes of a single bit – can have the most drastic consequences."*. Viewed under the perspective of statics, it means that Dijkstra considers software as a system in unstable equilibrium, or to put it more precisely, that the correctness of a program output is unstable with respect to perturbations. However, some previous works (see section 7) suggest the opposite, i.e., suggest that programs can accommodate perturbations.

In this work, our goal is to empirically verify this hypothesis. We devise an experimental protocol, ATTRACT, to study the stability of program correctness under execution perturbation. Our protocol consists in perturbing the execution of programs according to a perturbation model and observing whether it has an impact on the correctness of the output. We shall observe two different outcomes: the perturbation breaks the computation and results in an incorrect output, or the correctness of the output is stable despite the perturbation.

When a perturbation does not break output correctness, we observe "stable correctness", equivalent to stable equilibrium in statics. In such a case, we say that there is a phenomenon of "correctness attraction", an expression inspired by the concept of "attraction basin" in physics, which refers to the basin at the left hand-side of Figure 1, or more generally to the input points for which a dynamic system eventually reaches the same fixed and stable point.

We apply protocol ATTRACT to analyze 10 Java programs ranging from 42 to 568 lines of code in two separate perturbation campaigns called PONE and PBOOL. The comprehensive exploration of the perturbation space results in 2917701 separate executions. Among those 2917701 perturbed executions, 1977199 (67.76 %) yields a correct output. This experiment suggests that Dijkstra was wrong, and that there is a phenomenon of correctness attraction in software. We further validate this finding in a third experiment, PREAL, where we perturb the execution of two real-world large software applications (*bitcoin* and *bittorrent*). The results of PREAL confirm the presence of correctness attraction. Nonetheless, we note that those results are completely novel and that they are subject to bugs. Consequently, further experiments are required to validate them and to deepen our understanding of this intriguing phenomenon.

This document is organized as follow: First we will expose our concepts of perturbation, correctness ratio and give an example. Then we will explain two of exhaustive explorations we lead: PONE and PBOOL. We will address attention to two other subjects to explore the perturbability of production software: PREAL. Then, we present first work on next steps: PCOMMERCE. We will manually analysis at least 2 points per subjects to deepen understand our results. Finally, we will present some related works and conclude on this exploration of perturbability of Software.

1 The ATTRACT Protocol

1.1 Definitions

We consider programs that take inputs and produce outputs. Each input can be characterized by an input model, for instance "all arrays of integers". The **correctness** of the output is checked by an **oracle** [2]. The oracle is either a predicate on the output or a comparison of the output against the one produced by a reference implementation.

An oracle is said partial when it only checks for one aspect of the correctness: for example, if the program returns an array, an oracle that only checks the length of the array is partial. An oracle is said perfect – a **perfect oracle** – when it fully assesses the correctness of all aspects of the output: for instance, a perfect oracle for a sorting algorithm asserts that the output is sorted and that all elements of the input array are still in the output array.

In this work, an **execution** is a pair (program, input). An **execution perturbation** is a runtime change of the value of one variable in a statement or an expression. The change is performed only once per execution. Because the perturbation occurs at runtime, a perturbation has 3 dimensions: *time*: when the change occurs (e.g. at the fourth iteration of function 'foo'), *location*: where in the code (e.g. on variable 'i') and *kind*: what is this change, according to the type of the *location* (e.g. +1 on an integer value). Those 3 dimensions form a **perturbation model**. For instance, one can perturb integer variables by adding one to the value in memory. The **perturbation space** according to a perturbation model consists of all possible unique **perturbed execution** for a given program and input.

Correctness attraction is the phenomenon by which the correctness of an output is not impacted by perturbed execution, that is one can perturb an execution while keeping the output correct according to a perfect oracle.

1.2 Realization

To actually perform perturbations, we add **perturbation points** to the program under study where a **perturbation point** (denoted *pp*) is an expression of a given data type compatible with the perturbation model. For instance, if one perturbs integer values, the **perturbation point** will be all integer expressions (literal as well as compound expressions). In Listing 1[1], there are 3 potential integer perturbations points in a single statement, highlighted with braces.

$$\texttt{acc} \mid= \underbrace{i}_{pp_1} >> \underbrace{mask}_{pp_2};$$

$$\texttt{acc} \mid= \underbrace{i >> mask}_{pp_3};$$

Listing 1: Three integer **perturbation points** in a single statement.

In ATTRACT, we detect **perturbation points** with static analysis. We automatically add perturbation code around them using a code transformation. The transformation consists of wrapping all compatible expressions by a function call p (for "perturb")[2].

```
acc |= p(3, p(1,i) >> p(2,mask));
```

Listing 2: With perturbation code injected.

In Listing 2, each integer expression from Listing 1 program is wrapped in a call to function p. If p is the identity function, the transformed program is semantically equivalent to the original program. In order to only perturb one location at a time, the perturbation function p must know which expression is being perturbed. For that purpose, the function p takes an integer as parameter which uniquely identifies the **perturbation point** (from 1 to 3 in the example corresponding to the index given in Listing 1 under perturbation point *pp*).

[1] $\mid$ is the bitwise or operator. $>>$ is the binary right shift operator. The statement $acc \mid= i >> mask$ is equivalent to $acc = acc \mid (i >> mask)$.

[2] In our experiments, we implement this transformation on Java programs using the Spoon transformation library [3]

1.3 Perturbation Space Example

Let us give the perturbation space for a more complex example shown in Listing 3, inspired from real code. Only one perturbation point is considered, when reading the value of i in the loop statement.

```
int function(int bound) {
    int acc = 0;
    int mask = 0x2;
    for (int i = bound; i > 0; i--) {
        acc |= p(1, i) >> mask;
    }
    return acc;
}
```

Listing 3: Example of the instrumented function with the perturbation space reduce to 1 perturbation point

Consider the input $bound = 8$. After a first run of the program without any perturbation, one knows that perturbation point 1 is executed 8 times (the loop is bounded by the input *bound*). Table 1 shows the trace of the perturbation-free execution – the reference run – for values *acc* and i. The circled value is the final output returned by the function: the value of variable *acc*.

Table 1

It.	*acc*	i
1	2	8
2	3	7
3	3	6
4	3	5
5	3	4
6	3	3
7	3	2
8	③	1

Then, we perform a systematic exploration of all possible perturbations. For the example, the perturbation model consists of incrementing integer expressions (called the PONE model). The traces of the perturbed executions are shown in Table 2, where a **+1** means that the perturbation has occurred at the given iteration. First, the exploration reveals the fact that the first three executions have the same final output as the final output of the reference run: 3. In the second execution the value *acc* is impacted by the perturbation, *i.e.* it is not equals to the value of the reference run at the same iteration (the red value). But at the next iteration, the computation makes it slightly converge to the right value (the green value), and the final output is indeed correct. For this example, there is a phenomenon of "**correctness attraction**".

Table 2: Trace of three executions. The **+1** indicates when the perturbation on the value of i occurs. The red value of *acc* indicates that the perturbation on i has an impact on *acc* compared to the reference run. The green values signals that the value of *acc* converge to the right value.

it.	*acc*	i	*acc*	i	*acc*	i
1	2	8 **+1**	2	8	2	8
2	3	7	2	7 **+1**	3	7
3	3	6	3	6	3	6 **+1**
4	3	5	3	5	3	5
5	3	4	3	4	3	4
6	3	3	3	3	3	3
7	3	2	3	2	3	2
8	③	1	③	1	③	1

Let us now consider more inputs, and not only $bound = 8$ and perform the exploration of the perturbation space for *bound* ranging from 0 to 100. We find out that over the 4950 executions required to explore the space, there is only 5 failures (the final output of the perturbed execution is different from the output of the reference run one): $bound = 3$ at $It = 0$; $bound = 7$ at $It = 0$; $bound = 15$ at $It = 0$; $bound = 31$ at $It = 0$; $bound = 63$ at $It = 0$; This means that only 0.10% of the perturbation space yields an incorrect output. The 99.90% of perturbed yet correct executions indicates a high presence of **correctness attraction** in this code snippet for this particular perturbation point.

Let us discuss one the 5 incorrect outputs due to perturbation, when $bound = 3$ and when the perturbation occurs at the first iteration. Table 3 shows the corresponding execution trace. The trace clearly shows that the computation does not converge to a correct output.

Table 3

it.	*acc*	*i*
1	1	3 **+1**
2	1	2
3	①	1

1.4 Core Algorithm

We now formalize the example given in subsection 1.3 and present Algorithm 1 for exploring the perturbability of a program. It is based on the concept of perturbation points as presented in subsection 1.2. This algorithm requires: 1) a program 2) a perturbation model 3) a set of inputs 4) a perfect oracle.

The code design goal of this algorithm is to systematically explore the perturbation space. Algorithm 1 first records the number of executions of each perturbation point for each input– $n[pp, i]$ – without injecting any perturbation[3] Then, it re-executes the program for each input, with a perturbation for all points so that each point is perturbed at least and at most once per input. A perturbed execution can have three outcomes: a success, meaning that the correctness oracle validates the output; a failure meaning that the correctness oracle is violated (also called an oracle-broken execution) ; a runtime error (an exception in Java) meaning that the perturbation has produced a state for which the computation becomes impossible at some point (*e.g.* a division-by-zero). This algorithm performs a systematic exploration of the perturbation space for a given program and a set of inputs according to a perturbation model.

Input:
prog: program,
I: set of inputs for program *prog*,
oracle: a perfect oracle for program *prog*
Output:
exc: counters of execution per perturbation point,
s: counters of success per perturbation point,
ob: counters of oracle broken per perturbation point

```
instrument(prog)
for each input i in I do
  n[pp, i] ← runWithoutPerturbation(prog, i) ∀pp ∈ prog
  for each perturbation point pp in prog do
    for j = 0, to n[pp, i] do
      o ← runWithPerturbationAt(prog, i, pp, j)
      if catch Exception then
        exc[pp] ← exc[pp] + 1
      else if oracle.assert(i, o) then
        s[pp] ← s[pp] + 1
      else
        ob[pp] ← ob[pp] + 1
      end if
    end for
  end for
end for
```

Algorithm 1: Systematic Exploration of the Perturbation Space. The statement *runWithoutPerturbation(prog, i)* return the number of time each perturbation points are executed in the program *prog* for the given input i by running *prog* with i as input and logging each perturbation point. In the other hand, *runWithPerturbationAt(prog, i, pp, j)* runs the program *prog* while activating the perturbation point *pp*, at the execution number j for the given input i.

This algorithm enables us to compute:

[3] $n[pp, i]$ in the remaining of this work is the number of executions of perturbation point *pp* for a given input i. In fact, the number of execution of a perturbation point depends on the considered input, as we have seen in Listing 3 (where the number of execution of the perturbation point depends on the input *bound*).

- ϕ the proportion of correct executions per perturbation point ($s[pp]/\sum_i n[pp,i]$);
- χ the proportion of oracle-broken executions per perturbation point ($ob[pp]/\sum_i n[pp,i]$);
- ξ the proportion of exception-broken executions per perturbation point ($exc[pp]/\sum_i n[pp,i]$);
- Ω the total number of correct executions over the whole perturbation space ($\sum_{pp} s[pp]$).
- Φ the percentage of correct executions called **correctness ratio** over the whole perturbation space ($\Omega/\sum_{pp}\sum_i n[pp,i]$).

For a given program, if Φ is very low, Dijkstra's intuition is validated. If Φ is high, it means that the program under consideration exposes a kind of correctness attraction.

1.5 Correctness Attraction Modulo Input

Results of the systematic exploration of the perturbation space is sensitive to the input considered for each subjects. The "size" of the input considered has an impact on the computation and consequently to the exhaustive exploration. For instance, if a loop is iterating over a whole array given as input, the program may have a different behavior dependent to this number of iteration. In addition to this, the input distribution has also its own impact on computation.

Table 4: Dataset of 10 subjects programs used in our experiments. First column given the designation of the subject, second is the number of line of code of the implementation used. The third column briefly describe the task considered and the fourth column is the perfect oracle used. For more information about this subjects, see Appendix A

Subject	LOC	description	oracle
quicksort	42	sort an array of integer	output well sorted and with same values than input
zip	56	compress a string without loss	output equals to the input
sudoku	87	solve 9x9 sudoku grid	all constraint and input cell are respected
md5	91	hash a message 5	compare with the reference run
rsa	281	crypt with public and private keys	output equals to the input
rc4	146	crypt with symmetric key	compare with the reference run
canny	568	edge detector	compare with the reference run
lcs	43	compute the longuest common sequence	compare with the reference run
laguerre	440	find roots for polynomial functions	compute the equation with the root and compare to 10^{-6}
linreg	188	compute the linear regression from set of points	compare with the reference run

2 The PONE Experiment

We now present the PONE experiment. Its goal is to explore correctness attraction according to increments (+1) of integer values at runtime.

2.1 Perturbation model

In the PONE experiment, we perturb integer expressions. The PONE perturbation model is one of the smallest possible perturbation on an integer expression: a single increment of an integer value only once during an execution. An equivalently small perturbation model is MONE consisting of decrementing integers. As we will see later subsection 2.4, it gives equivalent results.

2.2 Pilot experiment with QuickSort

We consider the implementation of *quicksort* exposed in section A. This implementation contains 41 integer expressions (variables, literals of compound expressions), they are all perturbation points for PONE.

We create 20 random inputs for *quicksort*: 20 random arrays of 100 integers between $-2^{31}-1$ and $2^{31}-1$ According to those inputs, the 41 perturbation points are executed between 840 and 9495 times per input, which results in a perturbation space over all 20 inputs of 151444 possible perturbations.

Table 5: The breakdown of correctness attraction per perturbation point in Quicksort for integer point.

IndexLoc	#Perturb. Execs	#Success	#Failure	#Exception	Correctness ratio
0	1751	1202	543	6	68%
1	1751	1641	0	110	93%
2	1751	1751	0	0	100%
3	1751	1751	0	0	100%
4	1751	1751	0	0	100%
5	1751	1751	0	0	100%
6	1751	1751	0	0	100%
7	1751	1751	0	0	100%
8	1751	1751	0	0	100%
9	1751	1739	0	12	99%
10	5938	5938	0	0	100%
11	5938	5938	0	0	100%
12	8459	5946	2496	17	70%
13	8459	8459	0	0	100%
14	8459	8442	0	17	99%
15	4272	4272	0	0	100%
16	9495	6676	2691	128	70%
17	9495	9477	0	18	99%
18	9495	9495	0	0	100%
19	5308	5308	0	0	100%
20	4187	4187	0	0	100%
21	4187	3616	571	0	86%
22	3616	105	3506	5	2%
23	3616	0	3564	52	0%
24	3616	3616	0	0	100%
25	3616	3616	0	0	100%
26	1751	1633	118	0	93%
27	1751	1751	0	0	100%
28	840	275	565	0	32%
29	840	840	0	0	100%
30	1751	1751	0	0	100%
31	1751	1632	119	0	93%
32	891	321	570	0	36%
33	891	801	0	90	89%
34	3616	2361	1250	5	65%
35	3616	0	3616	0	0%
36	3616	1515	2101	0	41%
37	3616	742	2822	52	20%
38	3616	553	3058	5	15%
39	3616	1420	2149	47	39%
40	3616	0	3616	0	0%

Does *quicksort* exhibit correctness attraction under integer perturbation? We have exhaustively explored the perturbation space according to the PONE perturbation model. Among the 151414 perturbed executions, 77% result in a perfectly correct output – a sorted arrays containing all input values in this case. This shows that *quicksort* is robust to perturbations. This result calls for further research questions: does this hold for other programs as well? Can we perturb other data types? Those research questions will be answered respectively in subsection 2.3 and section 3.

In *quickSort*, are all perturbation points equally perturbable under integer perturbation? The breakdown per perturbation point is given in Table 5. The first column is the unique

identifier of the perturbation point, the second is the number of executions in which a perturbation has occurred. The third, the fourth and the fifth columns are respectively the sum of the number of successes, oracle-broken executions and exceptions over all perturbed executions. Finally, the sixth and last column is the correctness ratio. We see that there is a great disparity in correctness ratio over perturbation points: 1) for three points, a single perturbation always break the output correctness (*e.g.* point #23, #35 and #40). We qualify this kind of point as **fragile** because a single perturbation at runtime breaks the whole computation. 2) some points can be systematically perturbed without any consequence on the final output of the program (*e.g.* perturbation point #2 which has a correctness ratio of 100%). We call this kind of point **antifragile** (by opposition to fragile). The remainder is in between. We will explore whether the first finding holds for other programs than *quicksort* in subsection 2.3 and we will perform a manual analysis of perturbation points to deepen our understanding of the second finding in section 6. Finally, we also see that the perturbation of a given perturbation point can result in both an invalid runtime state yielding an exception, and in an incorrect output as identified by the oracle (*e.g.* for point #39).

2.3 Generalization over 10 subjects

We now consider the PONE perturbation model over 9 others subject programs + quicksort, summarized in Table 11 and explained in appendix A.

Table 6: PONE Results. The correctness ratio may not correspond directly to the number of Antifragile and Robust expressions because it is computed over all executions. Some points are executed much more than others, as explained in subsection 1.2.

Subject	N_{pp}^{int}	|Search space|	# Fragile *pp*	#Robust *pp*	# Antifragile *pp*	correctness ratio
quicksort	41	151444	6	10	19	77.6 %
zip	19	38840	5	2	5	76.09 %
sudoku	89	98211	12	27	8	68.8 %
md5	164	237680	102	24	7	29.67 %
rsa	117	2576	55	8	32	54.97 %
rc4	115	165140	60	7	12	38.04 %
canny	450	616161	58	129	133	94.55 %
lcs	79	231786	10	47	13	89.93 %
laguerre	72	423454	15	24	15	90.64 %
linreg	75	543720	43	18	11	47.88 %
total	1221	2509012	366	296	255	66.817 %

Table 6 gives the results of the systematic exploration of the PONE perturbation space. For each subject, this table gives:

- the number of integer perturbation points N_{pp}^{int};
- the number of perturbed executions (equal to the size of the PONE perturbation space);
- the number of **fragile** integer expressions;
- the number of **robust** integer expressions (correctness ration $> 75\%$);
- the number of **antifragile** integer expressions;
- the **correctness ratio** (percentage of correct outputs) over all perturbed executions.

Do the subjects exhibit correctness attraction? As shown in Table 6, all subjects expose some level of correctness attraction. For instance, for *zip* with 20 inputs, the PONE systematic exploration comprises 38840 perturbed executions, of which 76% yield a correct output. The perturbability ranges from a minimum of 29% for *md5* to a maximum of 94% for *canny*. All programs are indeed perturbable according to PONE, and to a large degree. Recall, that we have built the benchmark with no prior analysis of its stability against perturbation, there is no positive

bias in those results. In section section 6, we will provide a detailed account on the reasons for correctness attractions.

Are all perturbation points equally perturbable? As shown in Table 6, all programs contain **antifragile** integer expressions. This ranges from 7 points for *md5* to 133 point for *canny*. Similarly, the number of robust integer expressions (**correctness ratio** $> 75\%$) varies, raising as up as 129 points over 450 integer perturbation points for *canny* ($\approx 28\%$ of all PONE peturbation points).

We now study the breakdown of perturbability per integer expression in the code. Figure 2 gives the distribution of the correctness ratio per perturbed expression as a violin boxplot. Each line represents a program, the violin represents the distribution. The white dot is the median of the distribution. For instance, the first row represents *quicksort*, and the fact that most of the distribution mass is at the right hand side reflects that most integer expressions are perturbable. For *quicksort*, this is a visual representation of the data shown in the last column of Table 5. We call this violin distribution the "perturbability profile" of a program.

We interpret Figure 2 as follows. First, there is no unique perturbability profile: for *quicksort*, most integer expressions are perturbable (first row, the mass is at the right hand side, around 100% of success); for *md5*, most integer expressions are not perturbable (fourth row, the mass is at the left hand side, around 0% of success); for *linreg*, there are two groups of points, fragile ones with low correctness ratio and antifragile ones with high correctness ratio. Second, this distribution is coherent with the large proportion of perturbed-yet-correct executions reported in Table 6. Since there is a large number of robust and antifragile integer expressions, they account for a large share of the PONE search space.

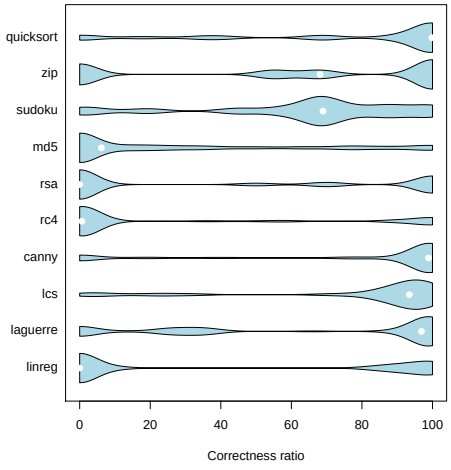

Figure 2: Perturbability profiles for all subjects

To sum up, the results of the PONE experiment show that:

- The considered programs are perturbable according to the PONE perturbation model;
- There are very few fully fragile integer expressions in the considered programs;
- There is a majority of highly perturbable integer expressions which results in a high level of correctness attraction.
- Dijkstra's view that software is fragile is not always true, correctness is rather a stable equilibrium than an unstable one.

2.4 The MONE Experiment

A natural small integer perturbation is to decrement an integer expression, we call this perturbation model MONE (Minus ONE). We have performed the systematic exploration of MONE. Table 12 in Appendix B gives the corresponding results. The results are really close to the results of PONE, and the presence of correctness attraction is similar. For instance, for *quicksort*, the correctness ratio under perturbation is 76,90% for MONE, very close to the 77.60% of PONE.

3 The PBOOL Experiment

Now we consider another perturbation model: instead of perturbing integer expressions, we set up an experiment to perturb boolean expressions.

3.1 Perturbation model

The PBOOL perturbation model perturbs boolean expressions. PBOOL perturb a boolean expression as follows: flip the value from *true* to *false* and vice-versa, i.e., computes the negation of the original value.

An interesting feature of PBOOL is that it often results in changing the control flow, since boolean expressions acting as if or loop conditions are also perturbed. To this extent, a PBOOL perturbation can be considered larger than a PONE perturbation studied in section 2.

3.2 Example on *quicksort*

We consider the same implementation of *quicksort* as presented in section A and perform a systematic exploration of the perturbation space for the same 20 inputs.

Does Quicksort exhibit correctness attraction under boolean perturbation? We have exhaustively explored the perturbation space of *quicksort* for 20 inputs according to the PBOOL perturbation model. This results in 31581 perturbed executions, of which 47% result in a perfectly correct output. This is less than for PONE but still high. Since all perturbations necessarily result in a different control flow (since all boolean expressions are in control-flow branches in this implementation of *quicksort*), it shows that many different paths exist that result in a fully correct output.

In *quicksort*, are all perturbation points equally perturbable under boolean perturbation? Table 7 presents the results for all 6 boolean perturbation points in *quicksort*. As we can see, there is also a variety in correctness ratios which ranges from 15% for boolean expression #2 to 99% for boolean expression #0.

Table 7: The breakdown of correctness attraction per boolean perturbation point in *quicksort*.

IndexLoc	#Perturb. Execs	#Success	#Failure	#Exception	correctness ratio
0	5938	5932	0	6	———— 99%
1	8459	1779	6663	17	— 21%
2	9495	1486	7991	18	– 15%
3	4187	3616	571	0	———— 86%
4	1751	1087	664	0	——— 62%
5	1751	1072	679	0	——— 61%

3.3 Generalization over 10 subjects

We consider the same 10 subjects and oracles as for the PONE experiment. Table 8 gives the results of the PBOOL systematic exploration. For each subject program this table gives:

- the number of perturbation points;
- the number of perturbed executions (equal to the size of the PBOOL perturbation space);
- the number of fragile boolean expressions;
- the number of robust boolean expressions (correctness ratio $> 75\%$);
- the number of antifragile boolean expressions;
- the correctness ratio over all perturbed executions.

Do the subjects exhibit correctness attraction under boolean perturbation?

As shown in Table 8, 2 subjects (*zip* and *md5*) are completely broken under boolean perturbations, a single perturbation of a boolean expression breaks the output. For *rc4*, only 7% of outputs remain correct under boolean perturbations. On the other hand, for 7/10 subjects, the correctness ratio is high, from 16% to a maximum of 83% for *laguerre*. What we have observed for the small integer perturbation model PONE also holds for PBOOL: there exists multiple equivalent execution paths to achieve the same correct behavior. Although PBOOL perturbations are more radical than PONE perturbations, correctness attraction still occurs. In section section 6, we will discuss boolean expressions in details.

Table 8: The Results of the PBOOL Experiments.

Subject	N_{pp}^{bool}	|Search space|	# Fragile *pp*	#Robust *pp*	# Antifragile *pp*	Correctness ratio
quicksort	6	31581	2	2	-	47.41 %
zip	6	14280	5	-	-	0.78 %
sudoku	26	28908	14	9	1	52.8 %
md5	10	12580	9	-	-	0.95 %
rsa	20	620	7	2	5	49.68 %
rc4	7	10540	7	-	-	7.59 %
canny	79	77038	28	10	14	71.55 %
lcs	9	25165	6	1	-	55.13 %
laguerre	25	109837	8	11	-	83.81 %
linreg	15	98140	10	-	4	0.04 %
total	203	408689	96	35	24	36.974 %

Are all boolean perturbation points equally perturbable?

Figure 3 gives the distribution of the correctness ratio per perturbed boolean expression as a violin boxplot (the same visualization used and presented in **??**). Each line represents a program, the violin represents a mirrored distribution. The white dot is the median of the distribution. Following the terminology set in subsection 2.3, those violin distributions are the "boolean perturbability profile" of programs. Here, we can see that there is no unique perturbability profile: for *zip*, no boolean expression is perturbable; for *canny*, many boolean expressions are perturbable (the 7th row).

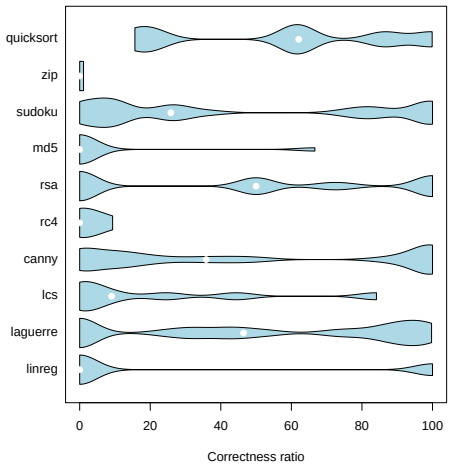

Figure 3: The distribution of PBOOL Perturbability as Violin Distribution Plots.

To sum up, the results of the PBOOL experiment show that:

- The observed perturbability of integer expressions also holds for boolean expressions: there is a correctness attraction phenomenon also with boolean value switching;
- There are fully fragile boolean expressions and fully fragile subjects in the considered subjects;

4 The PREAL Experiment

So far, we have considered small programs. Now, we would like to see whether we also find correctness attraction in programs of a larger scale. The PREAL experiment consists of perturbing the execution of a large scale application for which we are able to define a perfect oracle. We consider two such applications: 1) bitcoinj is a Java implementation of the bitcoin electronic cash system 2) ttorrent is a Java implementation of the peer-to-peer file sharing system.

4.1 Perturbation models

We use perturbation models PONE and PBOOL respectively presented in section 2 and section 3, *i.e.* we add one to integer value, and negate boolean one.

4.2 Challenges

The main challenge of PREAL is the size of the perturbation space. First, there are much more perturbation points in these applications. For instance, there are 8143 perturbable integer expressions in bitcoinj, which is 18x more than for the biggest of our previously used subjects (the canny edge detector which has 450 perturbable integer expressions). The second challenge is that some perturbation points are executed a very large number of times. For instance, the maximum for bitcoinj is an integer expression line 163 of Utils.java that is executed 855130 times for a single bitcoin transaction. *For real applications, the perturbation space is too large to exhaustively explored.*

Recall that the exhaustive exploration of the perturbation space requires to explore the Cartesian product of perturbation points by their iterations. In other words, the exhaustive exploration of the perturbation space of bitcoinj would take approx. 1786 days, that is 4.8 years[4] Consequently, in PREAL, we replace systematic exploration by random exploration within a time budget as explained above in subsection 4.3.

4.3 Protocol

The exploration protocol of PREAL shown in Algorithm 2. For a given time budget, we generate a random input, we randomly select a perturbation to be done, *i.e.* the perturbation point and its iteration number, then we run the task with one injected perturbation and finally we check the correctness of the execution according to the oracle (explained in subsubsection 4.4.1 and subsubsection 4.4.2).

4.4 Subjects

4.4.1 Bitcoin

Presentation Bitcoin is a payment system which allows users to do transactions using a virtual currency. We use an implementation called bitcoinj[5] in Java. This library allows one to build virtual wallets and perform transactions between them.
Size This implementation is composed of 243 java classes with 8143 integer perturbation points and 4952 Boolean ones.
Correctness Oracle We compare the state of the two wallets involved in a transaction at the end of transaction. The sender must have its initial amount minus the amount sent and the fee (fixed to 1 bitcoin for the sake of simplification). On the other hand, the receiver must have been credited the amount sent to its wallet. This is a perfect oracle.

If a perturbation results in an error or a failure, we run a recovery routine, in order to put bitcoinj wallets in a correct state: ready to receive and to send transactions. For the experiment, we build 3 fake bitcoinj wallets.

[4] An execution takes approx 4 seconds, and for exploring the whole space we need to do 10022748 executions
[5] `https://frama.link/TLZANX7F`

Input: prog: program
Output:exc, s, ob: counters of executions per perturbation point

$PP \leftarrow$ all pp of program prog
while $timeElapsed < timeBudget$ **do**
  $i \leftarrow$ generate random input
  $n[pp, i] \leftarrow runWithoutPerturbation(prog, i)\ \forall pp$
  $pp \leftarrow selectRandomly(PP)$
  $j \leftarrow selectRandomly(n[pp, i])$
  $o \leftarrow runWithPerturbationAt(prog, i, pp, j)$
  if catch $Exception$ **then**
    $exc[pp] \leftarrow exc[pp] + 1$
  else if $oracle.assert(i, o)$ **then**
    $s[pp] \leftarrow s[pp] + 1$
  else
    $ob[pp] \leftarrow ob[pp] + 1$
  end if
end while

Algorithm 2: PREAL: Random Exploration of the Perturbation Space

4.4.2 BitTorrent

Presentation BitTorrent is a protocol of peer-to-peer data sharing. We used a Java implementation called ttorrent[6] which allows us to create a torrent file from a file, share it and let it downloaded from another client. We generate a text file at random, containing random characters. Then we create the torrent file, and one node shares it from an input folder. An other node downloads the file into an output folder.
Size This implementation is made of 46 java classes, with 1256 integer perturbation points and 625 boolean perturbation points.
Correctness Oracle We compare the sent input file to the received file, bit by bit. The received file must be exactly the same one as the sent one. This is a perfect oracle.

4.5 Results

Results of the sampled exploration for *bitcoin* and *torrent* are summurized in Table 9. For 72 hours of time budget granted to the exploration for each subject, the exploration results in 26458 executions for *bitcoin* and 13741 execution for *torrent*.

Do large-scale subjects accept perturbation? The random exploration of perturbation space yields 19723 (74.54%) correct output for *bitcoin* and 11270 (82.01%) correct output for *torrent* for both **perturbation model**. Both of large-scale subject are perturbable according both **perturbation model** PONE and PBOOL.

Are all points are equally executed during the considered task? Over the 8143 integer perturbation point of *bitcoin*, only 2189 (26.88 %) and 1194 over the 4952 boolean one (24.11 %). On the other hand, the exploration is more exhaustive for *torrent* with 838 over 1256 integer perturbation points (66.71 %) and over 625 boolean perturbation 465 are explored(74.4 %). The coverage of the sampling exploration is quite low for *bitcoin* but is satisfactory *torrent*. On the other hand, there is a greater disparity in the distribution of execution of points for *torrent* with a maximum of 51 for perturbation for one point with a median of 10, while the median for *bitcoin* is 6 with a maximum of 27 perturbation per point.

Does the type of perturbation point has an impact on the correctness ratio? Integer perturbation point yields in 76% and in 85.8% of correctness ratio for respectively *bitcoin* and *torrent* while boolean one yields in 73.6% and in 78.5% for respectively *bitcoin* and *torrent*. However, in small subject we observe that PBOOL is more radical than PONE. In this large subject, we do not observe this taught behavior while **perturbation model** remains the same.

[6] `https://frama.link/OvgJwBf5`

Table 9: Results of the PREAL experiment

Bitcoin	
Time budget	72hours
# explored integer perturbation points	2189
# explored boolean perturbation points	1194
median of #perturbations per point	6
max of #perturbations per point	27
# of executions	26458
# of correct output	19723
integer correctness ratio	76%
boolean correctness ratio	73.6%
Bittorrent	
Time budget	72hours
# explored integer perturbation points	838
# explored boolean perturbation points	466
median of #perturbations per point	10
max of #perturbations per point	51
# of executions	13741
# of correct output	11270
integer correctness ratio	85.8%
boolean correctness ratio	78.5%

Such results suggest that production subjects accept largely perturbations. A first exploitation of this results is the explanation that programs contains a lot of bugs but which they do not impact the computation, until it reaches a specific internal state. Then, this high correctness ratio suggest there is a lot of "randomizable" points. Such points can increase the resiliency and the security of programs by breaking the determinism.

5 PCOMMERCE Experiment

We adapt the PREAL experiment to explore the perturbability of an e-commerce application. This section is the results of first steps of the exploration on an e-commerce application. This exploration need further experiments and analysis.

5.1 Perturbation Model

As for section 4, we use the combination of the perturbation model of PONE and PBOOL, *i.e.* we add one to integer value, and negate boolean one. The perturbation occurs only one time at the first execution of the given perturbation point.

5.2 Subject

5.2.1 Broadleaf Commerce

Broadleaf Commerce,or *BLC*[7], is an open-source e-commerce framework. There is three main components in this experiment: 1) the core framework developed by *BLC*; 2) the example *DemoSite*[8] provided by *BLC* itself that implements the framework; 3) a "client" component that is interacting with the e-commerce application.

We consider as input the client's request on the web site. As output, the http response provide by the *BLC* server.

Correctness Oracle: Our oracle is partial for this experiment. In fact, the *oracle* used assert if the code status of the response is OK, in other words, if it is different to 404 or 500. We are conscious that oracle is weak, and we are looking to build one stronger, even perfect.

5.3 Challenges

Challenges remain the same as section 4, but here we need the intervention of an user. We implement a bot that is browsing the website example to generate random request and simulate activities on the application. This bot is implemented with Selenium [9].

5.4 Protocol

As for *bitcoinj* and *ttorrent*, explore exhaustively the perturbation space of *BLC* is not conceivable, or at least we are not able to do it in a reasonable time. However, we devise the Algorithm 3 in order to explore a sample of the perturbation space.

This algorithm use the "history" of active perturbation point for a given request. For each request receive, it look in its history: it selects randomly among the previously active perturbation point, or over the whole perturbation points if the given request as not been already receive and there is no history available. It assert the results of the request with a perturbation, if the results of the request with a perturbation. Finally, it updates the history of the request.

We need to define a bound for the experiment. In our case, we take a number of request handled by the server. We do not use a budget time as for section 4 because some request does not execute the point to be perturbed. We prefer to ensure a certain amount of perturbed executions.

On the other hand, Selenium allows to simulate as much as we want client. We can in that way produce more request in less time, and explore faster than with a single client.

5.5 Results

We apply Algorithm 3 to the *DemoSite* of *BLC* with 5 "monkeys" that interacting randomly with the application. We grant 5000 perturbed request. We perturb only the module **core** of *BLC*, that is resulting with 6008 perturbation points: 2120 integers and 3888 booleans.

This exploration results with a total of 10553 request, with 5574 execution perturbation so there is 52,82% percentage of perturbed request. Results are summarized in Table 10

[7] `http://www.broadleafcommerce.com/`, `https://github.com/BroadleafCommerce/BroadleafCommerce`
[8] `https://github.com/BroadleafCommerce/DemoSite`
[9] `http://docs.seleniumhq.org/`

Input:
prog: program,
I: set of inputs for program *prog*,
oracle: a perfect oracle for program *prog*
Output:
exc: counters of execution per perturbation point,
s: counters of success per perturbation point,
ob: counters of oracle broken per perturbation point

```
Launch ECommerce Application
PP ← all pp of program prog
Launch Bot
while nbRequestDid < nbRequestBudget do
  i ← bot.doRequest()
  n[pp, i] ← historyOf(i)
  ActivPP ← ∀pp ∈ PP, n[pp, i] > 0
  if ActivPP.isEmpty then
    pp ← selectRandomly(PP)
  else
    pp ← selectRandomly(ActivePP)
  end if
  o ← runWithPerturbationAt(prog, i, pp, 0)
  if catch Exception then
    exc[pp] ← exc[pp] + 1
  else if oracle.assert(i, o) then
    s[pp] ← s[pp] + 1
  else
    ob[pp] ← ob[pp] + 1
  end if
  update historyOf(i)
end while
```

Algorithm 3: PCOMMERCE: Exploration of the perturbation space with history of active point for a given request made by the client bot.

Does an application of e-commerce accept perturbation? Over the 5574 perturbed request, there is 84,79% of correctness ratio. That is a very high correctness ratio, but the oracle used is very weak, and probably much more tolerant than a perfect one.

What is the proportion of explored perturbation points? The experiments explored 193 perturbation points, that is to say 3,21 % of the current space. This proportion is very low. Some request are not handled by bots. This can be an explanation of the high correctness ratio.

For the further research, we have to introduce perturbation points in the whole application. Devise "smarter" bots, that are able to reproduce a "real client" behavior, for instance add some items to its cart and completely end the order, instead of just randomly interact with the application in order to increase the number of perturbation points explored and so to reflect more accurately the true correctness ratio.

One major lead of improving this experiment is to set up a "shadow" of the e-commerce application. That is to say, sandbox a replication of the application, and copy the request made by real client on it. This provide the perfect oracle that we lack: we can compare the result of the shadow application with the production application without impact on users. See **??**.

# Total perturbation point	6008
# Perturbation point explored	193
% Perturbation point explored	–3,21%
# Executions	5574
# Success	4726
Correctness ratio	——— 84,79%
# Request	10553
Execution ratio	——— 52,82%

Table 10: Results of the random exploration of *DemoSite* of *BLC* with 5 bots that randomly interact with the e-commerce application.

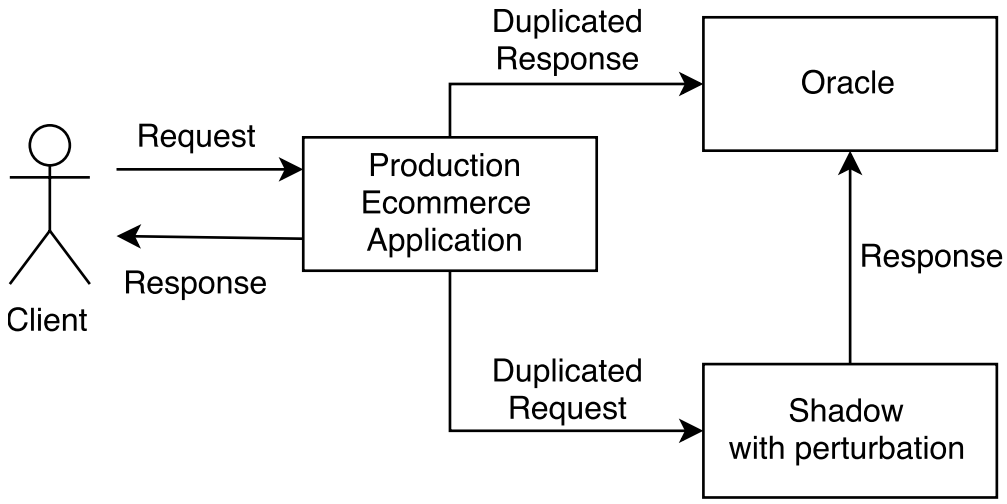

Figure 4: Concept of Shadow of e-commerce application where the client is not impacted by the exploration of perturbability envelop.

6 Manual Analysis

6.1 Methodology

For each subject and each experiment (PONE and PBOOL), we selected the perturbation points with the highest correctness ratio . Among this set of perturbation points, we selected ones with different behaviors, *i.e.* with different impacts on the computation. For each subject, we present a least one integer perturbation point and one boolean perturbation point.

6.2 Quicksort

Integer Location 8:

$$\textbf{int}\ \ \texttt{pivot}\ =\ \texttt{array}[\ \underbrace{beg + ((end - beg)/2)}_{\text{L8}}\ ];$$

The L8 expression in used to select a value of the input array as pivot. It has a correctness ratio of 100%, which means that one can always select another pivot and still obtain a correctly sorted array. This fact is known: whatever the pivot used, the algorithm is still able to successfully perform the sorting task. Actually, Moreover, one can randomly select the pivot, and this randomization breaks the *worst case execution time* (*WCET*) of Quicksort [**?**] (which is equivalent to the best practice of shuffling the array before running Quicksort [**?**]).

Boolean Location 0:

$$\textbf{while}\ (\ \underbrace{(left <= right)}\ )\ )\ \ \{\ \ \ldots\ \ \}$$

L0

The L0 expression controls a loop, we call it a emphloop control point. Under perturbation, either an iteration is added or the loop is stopped earlier. This perturbation point has 99.90% of correctness ratio. If the loop is stopped earlier than it should, there are more recursive calls, and the algorithm still converges to a correct state. The few failure happens one the perturbation adds an iteration.

Zip

Integer location 0:

```
int dictSize = 256 ;
               L0
```

This assignment it at the beginning of the decompress function of the LZW algorithm. By default, the algorithm has 256 entry to its dictionary. During compression, LZW adds 256 entries. With a PONE perturbation, $dictSize = 257$ while the actual size of the dictionary is 256 (from compression). Hence the 257^{th} entry remains empty. However, the rest of the algorithm never accesses this position, consequently, there are 100% of correct outputs after decompression.

However when we remove the last entry ($dictSize = 255$ under the MONE model), there are 6 tasks which fail. The reason is that they all contains the character ÿ which is encoded by the integer 255 (i.e. the last element of the dictionary), which is not added to the dictionary under MONE perturbation perturbation. When $dictSize = 255$, an access to the 256^{th} entry (at index 255) produces a failure.

Boolean location 4:

```
for (int i = 0; i < 256 ; i++)
                L4
    dictionary.put(i,""+(char)i);
```

The L4 expression controls a loop in method decompress too.

This location is the loop control point in method decompress too. It is the loop used to build the default dictionary, with the 256 characters already discussed. The perturbation stops the loop earlier or adds one iteration to it. When one iteration is added to the loop, it adds an extra yet useless entry to the dictionary. In the other hand, when the loop exits earlier, the dictionary lacks entries this results in a failure. This perturbation point has 1.09% of success, 56 success over 5140 perturbed executions.

Sudoku

Integer Location 78 and 79:

```
mBoxSubset = new boolean[mBoardSize][mBoardSize];
                         L78         L79
```

Expressions 78 and 79 are used in the initialization of the sudoku grid, *i.e.* mBoxSubset are the 9 boxes of 3x3 cells of the sudoku grids. The perturbation results with an extra row, column or box, which remain empty all along the computation of the solution because loops are bounded by the size of the input grid. The phenomenon of a perturbation adding extra resources to the program is a common one, we call it the *extra resource effect* and L78 and similar expressions are called *resources points* in the rest of this paper.

In contrast with PONE, an MONE perturbation on those locations results with 0 % of correctness ratio. As said, the loops are bounded by a fixed size, and the perturbation results in an out-of-bound access.

Boolean Location 97:

```
setSubsetValue(i, j, value, true);
                            L97
```

Expression L97 us also in the initialization code. It is used to set up the constraints on the input grid, to say which numbers are present, in which rows, columns and boxes of the Sudoku grid. The last boolean parameter is used to specify whether the cell is empty or not. When it is set to "false" (which is what the perturbation does), the parameter "value" is simply ignored. It means that the perturbation results in a different input grid with one cell that is empty, and considered by the Sudoku solver as such. By construction, if the problem is satisfiable, it is also satisfiable with an easier one, and it admits more solutions. Consequently, the perturbation yields a high correctness ratio of 99.67%. However, in come rare cases, the solution found is different from the original one and this is considered as a failure.

MD5

Integer Location 3:

```
int numBlocks = (((messageLenBytes + 8) >>> 6) ) + 1);
                 ^^^^^^^^^^^^^^^^^^^^^
                          L3
```

This integer expression is at the beginning of method "computeMD5()". The variable *numBlocks* is the number of blocks to be processed by the algorithm. When one perturbed L3 with MONE, the perturbation is nullified by the subsequent bitshift ($messageLenBytes + 9 >>> 6$ is equal to $messageLenBytes + 8 >>> 6$). With a greater perturbation (for instance adding 200 to the L3), the perturbation is not nullified and the correctness of the output is broken.

Boolean Location 22:

```
long messageLenBits = ((long) (messageLenBytes)) << 3;
for (int i = 0 ; i<8 ; i++) {
                  ^^^
                  L22
    paddingBytes[(((paddingBytes.length) - 8) + i)] = ((byte) (
        messageLenBits));
    messageLenBits >>>= 8;
}
```

This is the only boolean perturbation point that gives a correctness ratio greater than 0% in MD5. However, it yields a high 66% correctness ratio. For this a *loop control point*, the failure happens under perturbation when the loop does only one iteration or does one extraneous iteration. Interestingly, if the loop block is executed between 1-8 times, the result is correct. After analysis, the reason is that the other iterations are required for longer inputs. Since we only use short inputs of 100 characters, the useless iterations can be safely skips. [10] To verify our analysis, we a perturbed MDF with a string of 10000 characters and the computation wa indeed broken if the third and higher iterations are skipped due to the perturbation. What we observe is this case is that the correctness ratio overfits our input distribution model as explained in subsection 1.5 (*i.e.* strings of 100 characters).

RSA

Integer Location 0:

```
int bitSize = this.key.getModulus().bitLength();
              ^^^^^^^^^^^^^^^^^^^^
                       L0
if (forEncryption) {
    return (bitSize + 7) / 8;
} else {
    return (bitSize + 7) / 8 - 1;
}
```

[10] This loop sets the 8 last cells of the array with *messageLenBits*. This variable is initialized before the loop with the size of the array of byte to be left-shifted with 3. In our case, a string of 100 char is encoded in 148 bytes, and at the beginning $messageLenBits = 148 << 3 = 1184$. Then, at the second iteration $messageLenBits >>> 8 = 4$ and for all others iteration, $messageLenBits = 0$, that is why, when the loop stops after the second iteration it still works (because the array cells are already set to zero).

This statements collects in *bitSize* the number of bits that are required to encode the actual value of the modulus of the secret key. The L0 expression, a method call, has 100% correctness ratio with PONE. This value is used to assert if the given input is not too large to be ciphered by the current key. Our inputs are never too large to be processed, because we control the whole process of generation and their size is fixed to 100 bytes. The modulus is encoded on 1024 bits, so after the computation, the maximum size to be processed is $(1024 + 7)/8 = 129$ in case of encryption and $129 - 1 = 128$ in case of decryption. Moreover, in case of PONE, their is no change in the value returned by the function, that is why it is resulting with 100% correctness ratio.

But, this perturbation point could introduce a security issue. In order to cipher a large block with RSA, you need a large modulus. PONE does not produce any problem, because the state converge to the right value. However, in case of PMAG experiment, that is add a greater value than one to integer expression, the security is compromise. According to the PCKS#1[11] : "The length of the data D shall not be more than k-11 octets [...]", with k the length of the modulus which is at least 11 octets.

[11] `https://tools.ietf.org/html/rfc2313`: section 8 Encryption process

Boolean Location 48:

```
if ( output.length < getOutputBlockSize() ) {
                     L48
    byte[]  tmp = new byte[getOutputBlockSize()];
    System.arraycopy(output, 0, tmp, tmp.length - output.length, output
        .length);
    return tmp;
}
```

With PBOOL perturbations, we discover that perturbing this if-statement yields a 100% correctness ratio.

In this case, the perturbation flips false to true for 14 cases for one time, otherwise it is not executed. Which means that the code is not executed in nominal mode, and executed in perturbed mode. The then branch of this if statement copy the output array into a larger one in order to add a padding in the front of the output array. When the perturbation occurs, we know that the *output.length* is equals to *getOutputBlockSize*() or $tmp.length - output.length = 0$. Then the array is just copied, without any padding(because the offset = 0). This effects is the same that Wang *et al.* [4] observed and called "Y-branches" (see **??**). The alternate path and the original converge into the same final state.

RC4

RC4 is an encryption algorithm.

Integer Location 86:

```
for (int i=0; i < STATE_LENGTH; i++) {
              L86
    engineState[i] = (byte)i;
}
```

Location L86 is 100% PONE-perturbable. When a PONE perturbation occurs, the loop body is executed one time less (the last iteration) and consequently, the last cell of the array is initialized to 0 (default value of byte) instead of -1.

This integer perturbation point have 100% correctness ratio. This location is a *loop control point* as seen in Zip or MD5. This perturbation is used to initialize the engine state in *setKey()* method. The integer value of i is cast to a byte, values of engineState is in the interval $[-128; 127]$. The perturbation results with a 0 (non initialized) instead of -1 in the last cell of the array. The second step of the initialization, is swapping in the array in function of the key used. This array is then used to cipher / decipher blocks of bytes. The fact is, there is a swap, a xor and a mask (0xFF) on blocks and the wrong value doesn't impact the ciphering because of those operations.

Integer Location 114:

```
    i2 = ((keyBytes[i1] & 0xff) + engineState[i] + i2) & 0xff;
    byte tmp = engineState[i];
    engineState[i] = engineState[i2];
                     L114
    engineState[i2] = tmp;
    i1 = (i1+1) % keyBytes.length;
```

This location has 92,58%. It changes the value of one cell of the engineState array. This array contains unique element: integer on byte from -128 to 127. The perturbation result with a one value duplicated in it. As we said, values of this array are used to process block with binaries operations. Such operations seems to be insensitive to the fact that one value changed. In fact for MONE and PMAG we got the same correctness ratio.

Boolean Location 88:

```
for (int i=0; i < STATE_LENGTH ) ; i++)
             L88
```

This is a *loop control point* with 6,23% of correctness ratio. This is to loop of initialization (same loop than the location 86), but on the whole boolean expression. The loop is stopped earlier or do one more iteration than should. The extra iteration produce an error: out of bound of the array engineState. However, as we seen, the program still produce a correct output if we stop ealier: at the 237th iteration to the 255th. When the loop is stopped at the earliest possible, there is $255 - 237 = 18$ cells uninitialized, that to say equals to 0. We run a new exploration on the 114 (explained just above), in which we perturb 18 times per execution randomly. This experiment results with 38,83% of success. This is highlight the resiliency of the RC4 computation.

Canny

Integer Location 3:

$$\textbf{return } \underbrace{Math.round(0.299f * r + 0.587f * g + 0.114f * b)}_{\text{L3}};$$

This perturbation point has 100% of correctness ratio. It computes the luminance of pixel of the input image. The perturbation results with a stronger luminance for one given pixel, but after that, the filter normalize those luminance. After that, luminances are used by pack of 8: the gradients computation is based on neighborhood of the current pixel. The perturbation is dissolve in this computation.

Boolean Location 367:

$$\text{magnitude[index]} = \underbrace{gradMag >= MAGNITUDE\_LIMIT?}_{\text{L223}}$$
$$\text{MAGNITUDE\_MAX} : (\textbf{int}) \ (\text{MAGNITUDE\_SCALE} * \text{gradMag});$$

This perturbation points got 100% of success. This conditional is used in order to performs the "non-maximal suppression" of the canny filter. In our case, the expression $gradMag >= MAGNITUDE\_LIMIT$ is always false. So perturbation change the value of the magnitude from $(int)(MAGNITUDE\_SCALE * gradMag)$ to $MAGNITUDE\_MAX$. This is produce failures when the real value of magnitude is < 750 by default.For the 5 chosen input, this never happens. We run with more input for this precisely point and found out there is 10% of inputs that this location produce failures.

LCS

Integer Location 2 and 5:

$$\textbf{int}[][] \ \text{lengths} = \textbf{new int}[\underbrace{a.length() + 1}_{\text{L2}}][\underbrace{b.length() + 1}_{\text{L5}}];$$

These locations have 100%. As we can see, there is actually 6 perturbation that have the same results: adding one row or column to the lengths array (location from 0 to 5). These locations are *extra memory point* as we see in Sudoku for the locations 78 and 79. As we can expect, the MONE experiement results with 0% of correctness ratio while the PMAG gots 100% as the PONE. This array is used to compute to compute the distance between the strings.

Boolean Location 14:

$$\textbf{for } (\textbf{int } i = 0; \ i < a.length(); \ i{+}{+})$$
$$\textbf{for } (\textbf{int } j = 0; \ \underbrace{j < b.length()}_{\text{L14}}; \ j{+}{+})$$

This location has 44,36% of correctness ratio. It's a *control loop point*. The algorithm of LCS used implement dynamic programming: using that is already computed to compute the current value. Usually, these techniques induce one pre-computing of a array (here the array length) and then performs a back trace on this array(starting from the bottom of the array to the origin: cell 0,0 for instance). One of the usual optimization is to use a "k-band". This optimization eliminate some

values that we know they are worst than what we are looking for, the last cell of the first column for instance. This *control loop point* is building a kind of "k-band". Success happens if i is near to 0 (the left-bottom corner) and j near to $b.length()$ or when i is near $toa.length()$ and j near to 0 (the right-top corner).

Laguerre

Integer Location 73:

```
for (int j = n-1; j >= 0; j--) {
    d2v = dv.add(z.multiply(d2v));
    dv = pv.add(z.multiply(dv));
    pv = coefficients[ j ].add(z.multiply(pv));
                       L73
}
```

This location has 99,28 % of correctness ratio. This loops is in the method solve(), that is the core computation method. The perturbation result by using twice the same value (the previous one) and skipping the current value. The perturbation produce some errors: the algorithm can find solution the given problem. In others, the algorithm return something: a double with *Not a Number* value.

Boolean Location 0:

```
if (isSequence(min, z.getReal(), max)) {
                L0
    double tolerance = FastMath.max(getRelativeAccuracy() * z.abs(),
        getAbsoluteAccuracy());
    return (FastMath.abs(z.getImaginary()) <= tolerance) || (z.abs() <=
         getFunctionValueAccuracy());
}
```

This location has 96.15 % of success. It is in the function isRoot() that is check if the given value as arguments is weather a root (a solution) of the problem. First, if the perturbation occurs when the conditional is true, then the algorithm go thought the *else* branch, return *false* and do a new iteration: that is not a big deal because it will find a solution at this one. However, if it should false, then the algorithm computes an others condition, and reduce the amount of failures. Failures occurs only at the before last iteration, because the second condition is true for both the before last solution and the real solution, but not the first condition.

Regression

Integer Location 73:

```
Matrix x = new Matrix(a.getColumnDimension(), 1 );
                                              L73
```

This location has 100% of correctness ratio. The perturbation results an additional row to the Matrix x. This is an *extra memory point* as we seen in sudoku for instance. This point is in the method aTy: computation of the transpose of the Matrix a times the Matrix: $a^t \times y$. The Matrix x is the Matrix result of this computation. The program change the Matrix to an array, and iterate of the length of the given Matrix a then the extra row is not used. By running PMAG, whatever how many row we add, it still works. In the other hand, as we can expect, the MONE results with 0% of success, because the Matrix x has no row so it is empty.

Boolean Location 27 :

```
do {
    ...
    try {
        solution = ssWithRidge.solve(bb);
```

```
        for (int i = 0; i < nc; i++)
            m_Coefficients[i] = solution.get(i, 0);
        success = true;
                  L27
      } catch (Exception ex) {
        ridge *= 10;
        success = false;
      }
    } while (!success);
```

This location has 100% of correctness ratio. It is a *control loop point*. The perturbation results with an additional loop iteration. Instead of ending, it does a new call to the method solve(), and then end perfectly correct.

7 Related Work

Perturbation analysis of computer programs

Morell *et al* perform a perturbation analysis of computer program in [5]. The goal of this analysis is to evaluate the quality of a test strategy. They work on at the statement-level. Our injection of perturbation is similar to their own: introduction of method that takes the location and the original variable in arguments. As us, they perturb one time per execution, in the other hand, our perturbation model for integer is different: they change randomly bits of integer expression while we explored the impact of increment/decrement of this integer expression. However, the model for boolean expression is the same and we both perturb all the expression available in the program. If a test set has a higher propagation rate than other, then it should be consider better because it is using date state which is harder to hide errors in. They evaluated their perturbation analysis on 4 test strategy on 3 different programs and reveal that mutation test and data flow are better than statement and branch testing.

Hierarchical simulation approach to accurate fault modeling for system dependability evaluation

Zbigniew Kalbarczyk *et al* [6] devised a framework to inject physical fault into a simulator and trace and propagate its consequences until the system level. The method is iterative: each steps of the simulation is done one by one, with reporting the results of the previous one. For instance, The malfunction of the chip, produced by a bit-flip, create a memory corruption that has consequences on the next step: the system-level. They evaluated their methodology on Myrinet-based network system. They showed that their framework allows to propagate fault from the lowest level: physical to the system level: visible by the user. It also reduces the number of simulation needed with the dictionnary techniques and provide a valuable feedback for designing error recovery.

Decompilation as Search, Perturbation Analysis

In his Thesis Wei Ming Khoo [7] carried out a perturbation analysis to detect structural equivalent program. In fact, two structurally similar programs end with the same deviant state under perturbation for a given pair of variables. More such pair there are, more the two programs are structurally congruent. He ran this analysis on different implementation of sorting algorithm and on the *coreutils* tool set, with different compilation option. He compared two approach, the test based indexing and the perturbation based indexing. He then combined both to obtain following results on over 100 function of *coreutils* set: 68.2% of accuracy and 95.2% of recall.

Exploring the Acceptability Envelope

Rinard *et al.* [8] propose to consider the acceptable output it can produce to users. They claim that every perfect software are in a constellation of similar but imperfect system. Reaching the perfect software is (often) impossible, but it is surrounded by many system that the produce acceptable output: the “acceptability envelope”. Their injection of errors, perturbation for us, is to change every condition statement of for loop by adding or removing one iteration: change $q > expr$ to $q \geq expr$ for instance. This kind of errors injection is a subset of our *loop control point* explained in section 6. They explored the “acceptability envelope” for two application: pine a email client, and the Sure-Player MPEG decoder. The results on two real-world case studies show that while some errors are unacceptable, there are also some perturbations that are tolerated by the applications, hence providing opportunities for variant programs that all belong to the acceptability envelop. Off-by-one errors represent a specific case of the experiment PONE. Yet, our investigation of PONE is more extensive than the work by Rinard: we perturb more than loop conditions, and we exhaustively explore all possible perturbations. While Rinard *et al.* explored a subset of the constellation of the software called "acceptability envelop", we draw the entire constellation of perturbability one for the scenario considered. Our evaluation is different too, we use a perfect oracle and consider the output “correct” or “uncorrect”, while Rinard *et al.* must interact with the application because the “acceptability envelope” is user-dependent.

Automatic Mining of Functionally Equivalent Code Fragments via Random Testing

In [9], Jiang, Lingxiao and Su, Zhendong studied the *functionally* equivalent code with EqMiner, the tool they devised. They call *functionally* equivalent a specific case of *semantic* equivalent: it concerns the behavior of code from input and output standpoint. The fact is with a such approach, they save a lot of computation time. In order to lead this experiments, they extract piece of code from a program. This code has to compile in order to run it. They take as input variables that are used in it, and as output sides effect and outputs. They optimizing their experiments, with a random sampling of code fragments, parallelization and limiting of computation time for instance. They evaluates a sorting algorithm benchmark and the kernel linux 2.6.24. They compared their results with another tool, named DECKARD, that is mine syntactically equivalent code and found out there is more *functionally* equivalent code than syntactically: 68% of the code fragment extracted are functionally equivalent, and 58% of this code is structurally different. The random generation of inputs could be seen as perturbation of code fragments as we do, but they do not interested in studying the impact of such state change.

Perturbing and evaluating numerical programs without recompilation—the wonglediff way

In [10], P. R. Eggert and D. S. Parker presents wonglediff. This tool is built in two part: wongle, that creates a supervisor in parallel and allow to change the stage of process during their execution and numdiff, that produces an output resulting of the changes that wongle did. Wonglediff is used to change the rounding modes of floating-point number used by the program but its applications could multiple. The principle advantage of this tool compare to exist one, it is not required to recompile, re-link or reconfigure the process used. In fact, if this executable is able to run on Unix or linux kernel, so management process can change hardware register. They used wonglediff to determine the portability of an application because the floating-point rounding mode can change from a machine to another. Wongle build multiple sub-processes that run the application and change frequently the round method. Then numdiff produce an output and allow to analyze it. They evaluated it with a process that performs Strassen's recursive algorithm, they change the round method for this process and evaluate the accuracy of the computation. The perturbation and the exploration of its space are totally different that what we do. While we do an exhaustive exploration of the perturbation space of the state of process, they explored randomly the perturbation space of the floating-point rounding modes.

Y-Branches: When You Come to a Fork in the Road, Take It

Wang *et al.* [4] studied the reconvergence of the state of system. They instrumented a sample(1000) of conditional branch to force it to take the alternate path (*i.e.* take the *else* path when it should take the *then* one and vice-versa) and this forcing is done only one time per execution, as we do, in order to have independent forced switch. Their study is a specific case of our PBOOL experiment because they limited it to a random sample of conditional and only to conditional, not to all boolean expression. They look after branch that give correct output even if the alternate path is force. They called such branch Y-Branch and highlight the fact that software have correctness attraction point. They evaluate Y-branch on a large benchmark of 13 subjects. They studied two kind of reconvergence: control reconvergence and full state reconvergence. The control reconverge is when the two fork (instrumented and original) execute same instructions for the remainder of the program. In the other hand the full state reconvergence is when both execution reach the same architectural state. They propose to use Y-branch to avoid misprediction of branch.

Application-level correctness and its impact on fault tolerance

In [11], X. Li and D. Yeung studied the fault resiliency of program at application level instead of architectural level. In contrast with us, they evaluates only on application with multiple valid output while we explicitly choose the opposite: only application with perfect oracle. In the other hand, as us, they only evaluates on the final output of the computation. To test if an output is "acceptable" to a user, they compute a quality of this output by comparing it to a baseline

of free-fault executions. The threshold of fidelity acceptable by the user is user-dependent and application-dependent. In fact, the quality required can change from an user to an other. This evaluation is limited and can not be automated because of the intervention of the user. The fault injected (perturbation for us), is a single bit-flip. This perturbation are injected into 3 hardware structure: the physical register file, the fetch queue and the issue queue. After their experimentation, they conclude that 46% of incorrect architectural execution produce an acceptable output for the user over 6 multimedia and AI benchmarks. They propose recovery mechanism: they only put checkpoint at the program counter (PC) and save the register file and the stack. If a crash occurs, they rollback to the last checkpoint. They estimated the run time overhead at 1%. This lightweight recovery mechanism recover successfully between 3.8% to 17.7% on average for the multimedia and AI benchmark, and in 22.5% to 31.0% of recoveries on average for the SPEC benchmarks.

Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults

Siva Kumar Sastry Hari *et al* [12] studied the presence of silent data corruption(SDC). They carried out an exploration of the fault injection into hardware, as we do in software. However, our will is to explore it exhaustively while they devise a efficient techniques to prune fault injection location and execution, what they called number of hardware sites and application execution sites. Relyzer, the tools they present is able to prune 99.78% in 12 applications. Relyzer use two techniques: *known-outcome* and *equivalence-based*. The former is about to predicting the outcome of a fault, while the latter is about to show that a fault is equivalent to others using static and dynamic analyses and heuristics.

Conclusion

We devised a protocol and a tool, that allows to explore exhaustively the perturbation space of a given program, according to an input distribution and a perturbation model.

We explored successfully the whole perturbation space of 10 subjects and figure out they all accept perturbation and confirm the presence of correctness attraction bassin according to two perturbation model: PONE and PBOOL.

We carried out 2 others systematic exploration: PMAG and MONE. The MONE, stand for Minus ONE, experiment show save profiles of perturbation than MONE, while the PMAG, stand for Plus MAGnitude, *i.e.* a larger value than 1, experiment need further analysis and experiments.

In addition to this, we explored an active subset of the perturbation space of two large subject: bitcoin and torrent. We show such large program can also accommodate perturbation, more than we could except.

This experiments results with the falsification of the assumption that all program states are in an unstable equilibrium. We analyze deeply our 10 subjects in order to understand why those programs are naturally resilient to such perturbation.

This experiments inject only one time a perturbation. PMAG is way to increase the "strength" of the perturbation. However, an other way is to increase the "frequency" of perturbation. Perturb the same point more than one time per execution, with a random probability to activate the perturbation for instance. We call this experiment PRND.

This protocol can be used to find randomization point. This kind of point are very valuable in term of security in order to break the determinism of programs for instance. Randomization offer security robustness and is largely used, especially in cryptography algorithm or operating system.

We addressee the perturbability only to integer and boolean expression. **Could we observe same results on floating-point expression? on Java Object? Which perturbation model could be applied on this kind of expression?**

As Rinard *et al* described, the "perfect software" is surrounded by a constellation of imperfect, but acceptable software. We showed those program acceptable perturbations and can produce the correct output according them. That might be the explanation that there is a lot of bugs among software, but there are able to provide correct output, until it is under a state, or a given input, that the bug is too deeply in it and the program can not recover to an equilibrium itself and need a manual fixing from developers.

References

[1] Edsger W. Dijkstra. On the cruelty of really teaching computing science. December 1988.

[2] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[3] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, page na, 2015.

[4] N. Wang, M. Fertig, and Sanjay Patel. Y-branches: when you come to a fork in the road, take it. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 56–66, Sept 2003.

[5] L. Morell, B. Murrill, and R. Rand. Perturbation analysis of computer programs. In *Computer Assurance, 1997. COMPASS '97. Are We Making Progress Towards Computer Assurance? Proceedings of the 12th Annual Conference on*, pages 77–87, Jun 1997.

[6] Zbigniew Kalbarczyk, Ravishankar K Iyer, Gregory L Ries, Jaqdish U Patel, Myeong S Lee, and Yuxiao Xiao. Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. *IEEE Transactions on Software Engineering*, 25(5):619–632, 1999.

[7] Wei Ming Khoo. Perturbation analysis. *Decompilation as Search*, Chapter 5:65–79, August 2013.

[8] Martin Rinard, Cristian Cadar, and Huu Hai Nguyen. Exploring the acceptability envelope. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 21–30, New York, NY, USA, 2005. ACM.

[9] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92, New York, NY, USA, 2009. ACM.

[10] P. R. Eggert and D. S. Parker. Perturbing and evaluating numerical programs without recompilation—the wonglediff way. *Software: Practice and Experience*, 35(4):313–322, 2005.

[11] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 181–192, Feb 2007.

[12] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. *SIGPLAN Not.*, 47(4):123–134, March 2012.

[13] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

A Experiment Subject

Overview

In this works, we consider 10 java programs. We selected them to have a large panel of different kind of computation, and tried to always have a perfect oracle. The Table 11 is constructed as follow: 1st column is the name used to refer to the subject and the number of Line of Code (LOC) in the second. Then in the 3rd, 4th and 5th respectively the number of integer perturbation point, the number of boolean perturbation point and the total number of perturbation point for each subject. In the last column, it a brief description the computation considered. In the last row, there is the sum over all subjects.

In following experiments, we will study *quicksort* subject deeply, and then generalize to the 9 others.

Quicksort

Let us consider an implementation of a Quicksort algorithm in Java[12].

Correctness Oracle: The oracle checks that the array is correctly sorted by comparing elements of the output, that is to say the next element has to be greater than the previous one, and checks if each element of the input belongs to the output.

Subject	LOC	N_{pp}^{int}	N_{pp}^{bool}	N_{pp}	description
quicksort	42	41	6	47	sort an array of integer
zip	56	19	6	25	compress a string without loss
sudoku	87	89	26	115	solve 9x9 sudoku grid
md5	91	164	10	174	hash a message 5
rsa	281	117	20	137	crypt with public and private keys
rc4	146	115	7	122	crypt with symmetric key
canny	568	450	79	529	edge detector
lcs	43	79	9	88	compute the longuest common sequence
laguerre	440	72	25	97	find roots for polynomial functions
linreg	188	75	15	90	compute the linear regression from set of points

Table 11: Dataset of 10 subjects programs used in our experiments

Zip

The Lempel-Ziv-Welch (LZW) [13] is a lossless data compression algorithm. We used it to compress/uncompress string. The implementation come from RosettaCode too, with 1 class and 2 methods : on for compress, and the other to uncompress. The implementation has 6 boolean perturbation point and 19 numerical perturbation point over 56 lines of code.

Correctness Oracle: The oracle assert if the output string is the same as the input string.

Sudoku

A Sudoku solver taken from RosettaCode. We provide as input a grid generated randomly. Some cell are already fill with values and empty cell contains a zero. There is 1 class of 87 lines of codes, 89 numerical perturbation point and 26 boolean perturbation point.

Correctness Oracle: The oracle assert if all constraints of Sudoku are satisfied, if there is no more zero in it and if every "full" cell of the input grid are still in the output grid.

[12]The original code is available at `https://frama.link/XGMArl34`, while the same instrumented code is available at `https://frama.link/cMnZj_Q6`

MD5

The Message Digest 5 (MD5), used to hash a string of a given size. We took the implementation from RosettaCode website. There is 1 class with 1 method, and 91 lines of codes. We introduced 164 numerical perturbation point, and 11 boolean.

Correctness Oracle: The oracle is that the output is the same as the one from the reference implementation.

RSA

A RSA cryptosystem designed by Ron Rivest, Adi Shamir, and Leonard Adleman []. The input is a random string of 64 bytes. This implementation was taken from bouncy-castle[13][14]. The project is composed of 1494 classes with a total of 241483 lines of code. We studied the RSACoreEngine class, which has 6 methods with 203 lines of codes, 73 numerical perturbation point and 19 boolean perturbation point.

Correctness Oracle: The oracle asserts that the output string is the same as the input.

RC4

RC4 is a stream cipher designed by Ron Rivest. This algorithm is fast and simple but multiple vulnerabilities has been discovered. The class RC4CoreEngine has 150 lines with 7 boolean perturbation point and 112 integer one.

Correctness Oracle: The oracle is that the output is the same as the one from the reference implementation.

Canny

Application of a canny filter on handwritten digit image, from the Mixed National Institute of Standards and Technology (MNIST) database. The canny filter is used to detect edges on an image. The edges detector implementation has been found on the Tom's Gibara web site. There is one 1 class with 568 lines with 450 integer points and 79 boolean perturbation point.

Correctness Oracle: The oracle asserts pixel by pixel if the output of the perturbed execution is equals to the unperturbed one.

LCS

Longest Common Sequence[15]. As input, it takes two string, and compute the longest common sequence. As input, we use RNA sequences of two plants : sativa and thaliana, extracted from the mature dataset of miRBase[16]. This implementation got 43 Lines with 9 Boolean perturbation point and 79 integer perturbations.

Correctness Oracle: The oracle is that the output is the same as the one of the reference implementation.

Laguerre

A Laguerre root finder implementation for polynomial equation. The implementation come from The Apache Commons Mathematics Library [17]. The class studied is the LaguerreSolver class which is 440 lines long and got 176 integers points and 25 booleans.

Correctness Oracle: The oracle checks if with the solution provided the equation is effectively equals to zero. Because of the computation on floating-point numbers, we floor the results of the equation with $x = root$ and accept it if his absolute value is less or equals than 10^{-6}.

[13] `https://www.bouncycastle.org/`
[14] `https://github.com/bcgit/bc-java`
[15] `https://frama.link/3ZxP5eBj`, dynamic programming version.
[16] `http://www.mirbase.org/ftp.shtml`
[17] version 3.6.1 : `https://frama.link/tQCYrZ2W`

Linreg

A Linear Regression using the Tikhonov regularization (also called ridge regression). We took the implementation form Weka Library [18]. The class LinearRegression has 188 lines of codes, with 75 integer perturbation points and 15 boolean one. In order to generate input, we generate coefficients to build a function, and then we compute the images of random points (x) by this function (y).

Correctness Oracle: It checks if the coefficients computed by the regression is equals to a reference run.

B MONE results

Table 12: The MONE systematic exploration.

Subject	N_{pp}	|Search space|	# Fragile exp	#Robust exp	# Antifragile exp	Correctness ratio
quicksort	41	37390	7	11	16	76.72 %
zip	19	38840	5	4	5	73.24 %
sudoku	89	98608	21	27	8	71.32 %
md5	164	237680	104	22	5	26.7 %
rsa	117	2576	57	3	34	53.42 %
rc4	115	165140	57	8	14	42.78 %
canny	450	616161	64	124	132	93.83 %
lcs	79	231786	17	45	7	93.75 %
laguerre	72	423454	18	25	9	91.14 %
linreg	75	547512	40	18	14	48.29 %
total	1221	2399147	390	287	244	67.119 %

C Studies on quicksort

Worst case execution time

The average time complexity of the *quicksort* algorithm is $O(n \log n)$. The **W**orst **C**ase **E**xecution **T**ime, *WCET*, is in $O(n^2)$ and happens when the program has to sort an array that is already sorted, with a pivot that has the value of the first cell of this array (*i.e* the smallest) or when it has to sort an array that is totally unsorted, *i.e.* an array that all elements are not in the right cell, with a pivot that has the value of the largest value of its elements. An usual technique to counter this *WCET*, is to randomize the choice of the value of the pivot.

To measure performance, we simply count the number of iteration of internal loop, *i.e.* the loop that looking for value to be swapped.

We run the quicksort algorithm in the *WCET*. First, the program runs without perturbation, and then we run it with a perturbation on the index used to choose a pivot: that to say, the first cells of the (sub-)array. We set up a *RandomEnactor*, *i.e.* the perturbation occurs with a given probability, *e.g.* 0.05 of probability or in other words 5% of chance that the perturbation occurs. We choose as **perturbation models** PONE, since this model offers 100% of correctness ratio on those location. (see section 6) Because this optimization is about the WCET, it should not degrade the average, *a random input array.* That is why we run the same experiment in such case. The Table 13 sum up the results over 100 arrays of 4000 integers.

The first columns is the case considered, the second and the third are respectively the number of loop iterations of internal loop during the process (absolute) of the reference and the execution perturbation. The fourth and the fifth columns are respectively the number of execution of the perturbation point and the number of perturbations and the last column is the "speedup" measured, that is to say, the difference of the perturbed execution with the reference one according to the number of iterations of internals loops measured.

[18] version 3.8.0 : `https://frama.link/fCjiqzk2`

Case	# It. Ref	# It. P	#Calls	# Perturbations	Speedup (%)
WCET	8042521	7626046	3818	182	5,18
Random	44521	44134	3973	190	0,87

Table 13: Results of experiments of the performances of the quicksort algorithm.

We can see that in the *WCET*, there is a "speedup" of 5,18%, and there is no loose of performance in the average case. This little study shows that randomization can improve performance. Even if this randomizable location was already known, we empirically verify it and use our perturbations framework to set it up.

Could we use perturbations to discover some optimization techniques?

Execution path

We studied the impact of the perturbation on the execution path of the quicksort algorithm. In order to lead this experiment, we record the number of unsorted pairs of value in the array. This allows to trace an execution path, and the evolution of the state of the array to be sorted.

For a given array of 500 integers, we trace executions path of the systematic exploration. This exploration results with 60694 executions.

In the Figure 5, the blue solid line is the execution path of the reference run, greens lines are executions perturbations that end with a success and red ones end with a failure or an error. The x-axis is

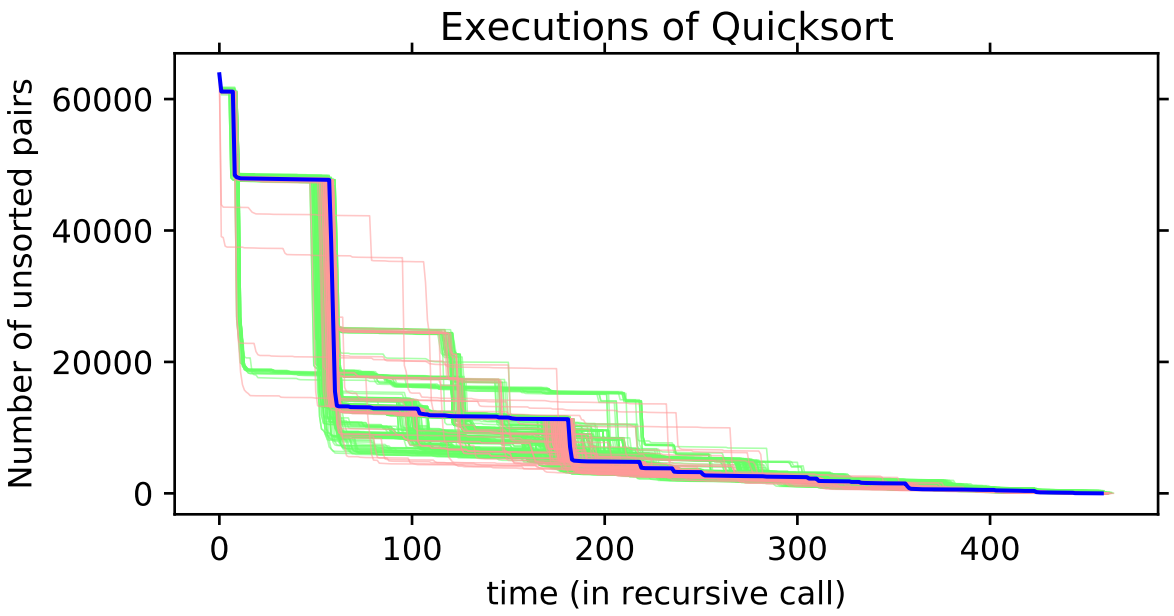

Figure 5: Executions paths of systematical exploration on 1 array of 500 integers.

We can see some of perturbed executions are faster than the reference run at the begin. However they all have the same "shape".

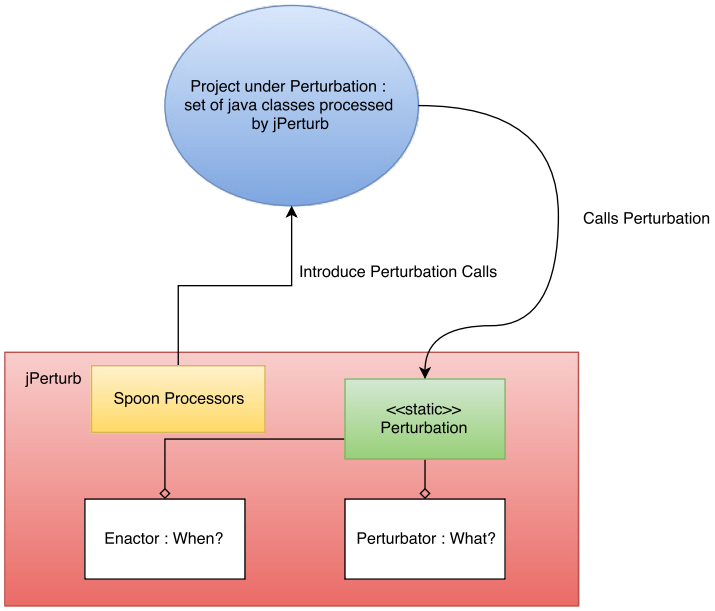

Figure 6: Diagram which show how jPerturb works

D Description of jPerturb

Overview

Perturbation Location

Perturbation Location, also called perturbation point, define the Where of the perturbation exploration.

Perturbation Models

Perturbators are the What to do. We can split (for the moment) them into two sub-group :

Numerical Perturbators

- PONE : return the given value plus 1.
- PMAG : return the given plus a magnitude.

Boolean Perturbators

- NEGB : return the negation of the given value.

Enactors

The When of the perturbation is defined by Enactors:

- N Execution Enactor : enact the perturbation point a the Nth execution of it. (example : N = 3, the point is enact at the 3rd execution)
- Random Enactor : enact the perturbation point with a given probability e. (example : e = 0.05, the point got 5% to be enact at each execution)
- always enactor: equivalent to a classical code mutation

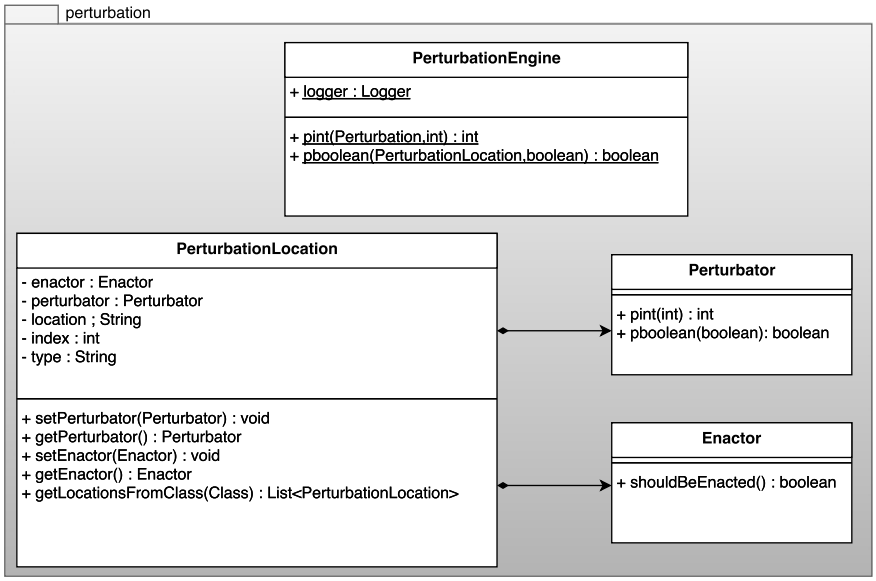

Figure 7: Class Diagram of jPerturb

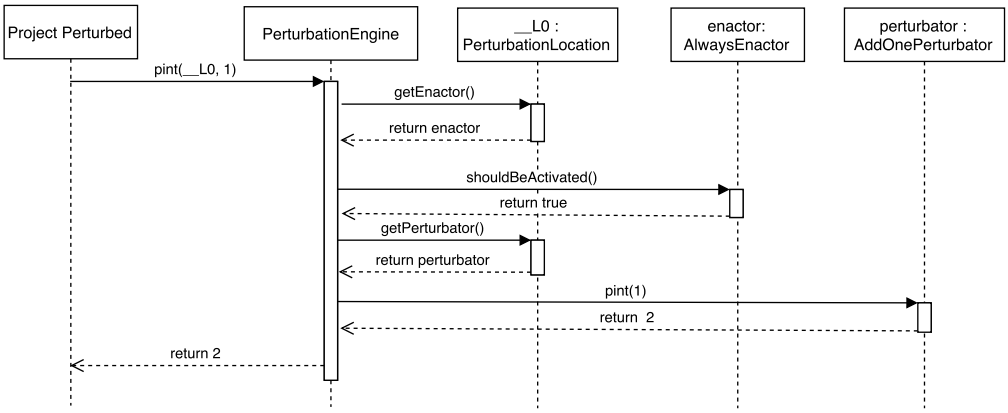

Figure 8: Sequences Diagram of jPerturb

Code

In the rest of th paper, perturbation in code will be referenced as follow :

```
pint(___LX, Y)
```

for integer perturbation and

```
pboolean(___LX, Y)
```

for boolean one. ___LX is the object Location used, with X the indice of the location. Y is the original value used by the program before instrumentation.