Mutation-Based Graph Inference for Fault Localization

Vincenzo Musco, Martin Monperrus, Philippe Preux

Mutation-Based Graph Inference for Fault Localization

Vincenzo Musco
University of Lille
Email: vincenzo.musco@inria.fr

Martin Monperrus
University of Lille
Email: martin.monperrus@univ-lille1.fr

Philippe Preux
University of Lille
philippe.preux@univ-lille3.fr

***Abstract*—We present a new fault localization algorithm, called Vautrin, built on an approximation of causality based on call graphs. The approximation of causality is done using software mutants. The key idea is that if a mutant is killed by a test, certain call graph edges within a path between the mutation point and the failing test are likely causal. We evaluate our approach on the fault localization benchmark by Steimann et al. totaling 5,836 faults. The causal graphs are extracted from 88,732 nodes connected by 119,531 edges. Vautrin improves the fault localization effectiveness for all subjects of the benchmark. Considering the wasted effort at the method level, a classical fault localization evaluation metric, the improvement ranges from 3% to 55%, with an average improvement of 14%.**

I. Introduction

Fault localization is the process of identifying the elements of software that are faulty, *i.e.* elements that are responsible of a bug. A classical way of stating the fault localization problem is that the bug is reproduced and asserted by a failing test case, and the goal is to predict the function that contains the buggy code. Apart from small and contrived examples, fault localization is not an error-less, fully analytic process. In a real software, fault localization does not diagnose faulty elements with certainty, it only predicts "suspicious elements" for which there are pieces of evidence that they may be faulty, and hence fault localization gives approximate and imperfect results.

In essence, the fault localization process tries to capture causality relationships between code elements [1], [2]. Indeed, early works in fault localization were based on program slices [3], which are refined versions of the most obvious causal relationship: the bug must lie somewhere in the code that has been executed. Spectrum-based fault localization is also causal to a certain extent, but with a really strong approximation: the causal relations are only captured by the fact that an element is covered by passing or failing test cases. That is, fault localization is only an approximation of the true cause-effect chain of error propagation that happens at run time. To our knowledge, only Baah et al. [1] and Shu et al. [2] have set notable milestones using causal inference for better approximating causal effects in fault localization.

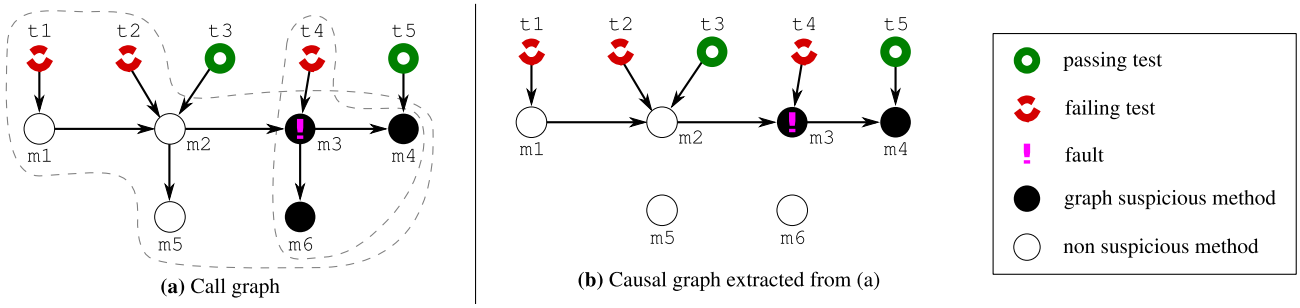
In this paper, we propose a novel approximation of causality for fault localization. Our insight is that call graphs are also approximations of cause-effect chains: if `A` calls `B`, a bug in `B` might result in a buggy output for `A`. We introduce Vautrin, a new fault localization algorithm that takes into account both call graphs and program spectra (*i.e.* execution traces of passing and failing tests [4]) to identify suspicious elements. Vautrin works in two phases: a "causal graph inference phase" during which mutants are used to track causality in call graphs. The idea is that if a mutant is killed by a test, there must be a path in the call graph between the mutation location and the test, along which the error has likely been propagated. Then, a "prediction phase" uses the graph-based causal knowledge to better identify the faulty elements.

To evaluate our algorithm, we consider the fault localization benchmark by Steimann et al. [5] published at ISSTA'13. We show that our method-level fault localization algorithm outperforms the most recent algorithms, including Ochiai [6] and Naish [7]. The improvement ranges from a minimum of 3% to 55% less methods to consider after fault localization. Also, we introduce a new evaluation metric which is the number of perfect predictions, that is, the number of cases in which the faulty element (a method in our case) is ranked at the top of the list, *i.e.* at position #1 in the suspicious list. Using our new technique, over the whole Steimann's dataset, the number of perfect predictions is 2,310 out of 5,836 which means a percentage of 40% representing an improvement of 14% over the related work.

To sum-up, this paper makes the following contributions:

- a new fault localization algorithm, called *Vautrin*, that uses a graph-based approximation of causality for fault localization;
- an empirical evaluation of Vautrin on 5,386 faults from the Steimann dataset, showing that Vautrin outperforms the state-of-the-art, both in terms of wasted effort and perfect fault localization prediction;
- a publicly-available implementation of our algorithm for Java.

The paper is structured as follows. In Section II, we present the concepts used in this paper and we introduce Vautrin, our fault localization algorithm. In Section III, we present the research questions under investigation, the empirical protocol and results, and provide answers to the research questions. In Section IV, we discuss the threat to validity of our work. In Section V, we present the related works. Finally, we conclude in Section VI.



(a) Call graph

(b) Causal graph extracted from (a)

Fig. 1. Example of (a) a call graph, and (b) the causal graph extracted from it using our technique. The causal graph only contains edges for which causal evidence exists.

II. Contribution

We introduce Vautrin, a novel fault localization approach working at the method level. Vautrin uses a call graph to build an approximate causal graph.

In this paper, a *causal graph* is defined as a directed graph in which nodes are all methods of a program. An edge is present in the causal graph between m_1 and m_2 if and only if there exists evidence that an incorrect state change may happen when m_1 calls m_2. The process of determining causal edges is called causal inference. In this approach, we note that nodes are never causal per se, only edges can be deemed as causal. This definition is not exactly that of Bayesian causal graphs [8] yet shares the same intuition. Our definition is tailored for the system, Vautrin, that we now present.

A. Introductory Example

Fault Localization is a field of software engineering which aims at automatically identifying a faulty element in a source code. A faulty element can be of any granularity (*e.g.* statement, method, class, *etc.*). In this paper, we focus on the method granularity, *i.e.* we propose a fault localization that predicts faulty methods. Most fault localization techniques are *spectrum-based* techniques where "spectrum" refers to the behavioral traces of the software when executing the tests [4]. Our technique uses another source of information, the call graph.

A call graph contains information about the methods and the way they call each others. Our intuition is that call graphs are an approximation of causality: if `A` calls `B`, a bug in `B` might result in a buggy output for `A`. Thus, Vautrin is built on the assumption that if two tests fail, the bug might lie at the intersection of the nodes reachable from both tests. However, this assumption does not always hold in practice because the ideal perfect causal graph is not known. A call graph is indeed only an approximation of cause-effect chains. The figure 1(a) illustrates a call graph for a simple program composed of 6 methods and 5 tests. Passing tests are nodes with a solid green border, and failing tests are nodes with a dashed red border. The faulty method is marked with an exclamation point. We compute the transitive closure of each failing test `t1`, `t2` and `t4`, that is the set of nodes reachable from each failing test. This transitive closure is represented with dashed lines on the figure. The intersection of the transitive closures gives a set of three nodes `m3`, `m4`, and `m6` (marked as plain black nodes), which contains the faulty method. In this paper, the methods in the intersection are called the *graph suspicious* methods (as opposed to spectrum-suspicious ones which are nodes determined using a spectrum-based fault localization algorithm). Conversely, other application nodes are marked with white nodes and are not suspicious.

In our example, method `m2` calls method `m5`. Let us imagine that method `m5` is a logging method. In such a case, a logging function is unlikely to propagate a fault as it only prints strings to a file. Thus, there is a difference between the causality flow and the method call flow. Some method calls may not alter the state of the system and thus should not be considered as causal. Moreover, a test may not call a method at run time. This occurs when the method invocation is optional, *e.g.* when located in a conditional block. In such cases, the causality of those edges should also be reconsidered. Indeed, the call graph contains all possible connections that are found by statically analyzing the source code[1]. Our idea is to only keep the call graph edges for which there exist pieces of evidence that they are causal.

For that purpose, we have a "causal graph inference phase" which consists in extracting a causal graph from the call graph. Figure 1(b) illustrates such a causal graph extracted from the call graph presented in Figure 1(a). This causal graph contains the edges of the call graph except the ones between `m2` and `m5`, and between `m3` and `m6` for which causal evidence does not exist (according to the process described in Section II-C1). This extraction process will be explained in Section II-B and Section II-C1. After the extraction, we observe that the set of faulty nodes based on graph analysis, contains one less node and still contains the faulty one.

B. Overview

We propose Vautrin, a new fault localization algorithm based on call graphs considered under the light of causality. Vautrin approximates causality by analyzing what happens between a failing test and a method based on the call graph.

[1]Some calls may be missed which are not statically observable, such as those resulting from the use of reflection.

When multiple methods are graph-suspicious, they are ranked using standard spectrum-based suspiciousness scores.

Vautrin uses a class hierarchy analysis (CHA) static call graph. This call graph is a directed graph which contains a node per software method and an edge from a method `A` to a method `B` if somewhere in the body of `A`, method `B` is called. This goes along the same line as the classical definitions as the one given in Grove et al. [9].

The causality is related to the propagation of an error in the program. One way to observe the propagation of a fault is to observe pairs of faults/failing tests: in Vautrin, they are used to build a causal graph. Our idea is to use mutation analysis for this. Mutation analysis [10] consists of assessing test suite quality by applying minor changes to a program resulting in what are called *software mutants*. A mutation in a program can be seen as simulating the insertion of a fault in the program. Thus, if we introduce a fault somewhere in the program, we can observe the impacts of this fault on the test cases. The propagation happens through method calls. Thus, a path in the call graph between a mutation point and a failing test is a potential path which has propagated the failure. Consequently, Vautrin is based on the assumption that mutants and their execution profiles do contain valuable information that can be used to approximate the causality. As explained in Section II-C1, Vautrin uses mutants to extract a causal graph out of a call graph.

Then, when a fault has to be localized, Vautrin computes the intersection of the transitive closure of each failing test according to the call graph. The nodes belonging to this intersection form the set of "graph-suspicious nodes". If the intersection is made of several nodes, Vautrin uses an external spectrum-based fault localization algorithm to rank them.

C. Algorithms

In this section, we present the two algorithms used by Vautrin for fault localization.

Vautrin uses software mutants to produce faulty executions. The mutants must be killed so as to observe (faulty location, failing test) case pairs. The first algorithm produces an approximate causal graph out of a call graph. To do so, Vautrin analyzes faulty executions of the program in order to identify which edges of the graph may be causal. To that end, Vautrin makes use of controlled faulty executions for which mutants involve certain tests to pass, some others to fail. From that information, Vautrin infers causal relationships, that is relationships like: given a set of failing tests, a set of passing tests, and a call graph, then the methods at the origin of the fault are likely to be this, and that. In Vautrin, edges are tagged as causal if they are part of a call graph path going from the failing test to a faulty method.

Once a causal graph has been inferred, the second algorithm is the prediction one: it is used at a production stage. This algorithm is executed at fault localization time within a debugging session by a developer. During this phase, Vautrin takes as input a set of failing tests and produces an ordered list of suspicious elements. The first element of this list is the most suspicious method, and the last element is the less suspicious method.

Algorithm 1: Building a causal graph from a call graph by approximating causality with mutants.

Input: G a call graph. I a set of pairs (mutant m, failing tests per mutant (T_f)).
Output: L: a causal graph

```
1 begin
2     L ← all nodes of G and no edge
3     for each {m, T_f(m)} ∈ I do
4         for each t ∈ T_f(m) do
5             add edges of shortest path from t to m in L
6     return L
```

1) Causal Graph Inference: The algorithm consists in inducing a causal graph from a call graph. A causal graph is a call graph which contains only the call graph edges for which causal evidence exists according to mutation analysis. This algorithm takes as input a set I of killed software mutants. Each mutant $m \in I$ lies in a method and a set of failing tests (T_f) that kills the mutant. If there exists a path between the mutated method and one of the failing tests in the call graph, the edges of the path are considered as potentially causal. Algorithm 1 describes this process.

On line 2, a new graph is created with all the nodes from the call graph and no edge. On line 3, we loop over each mutant m. For each failing test $t \in T_f(m)$ (line 4), we mark as causal all edges belonging to the shortest path going from the test to the mutated node method (line 5). At the end of the process, L contains the causal graph and is returned (line 6).

To illustrate this, let us explain how one can obtain the graph in Figure 1(b) starting from Figure 1(a). We assume that a first mutant in method `m3` results in 4 failing test cases: `t1`, `t2`, `t3` and `t4`. The algorithm will result in 4 paths: one from `t1` to `m3`, a second from `t2` to `m3`, a third from `t3` to `m3` and a last one from `t4` to `m3`. After processing this mutant, the causal graph contains 6 causal edges. Let us now imagine that we generate a second mutant occurring on `m4` resulting in two failing tests: `t4` and `t5`. The graph analysis of this second mutant results in two paths: the first from `t4` to `m4` and the second from `t5` to `m4`, this results in adding two more causal edges. With those two mutations, we obtain the causal graph shown in Figure 1(b).

The rationale of using the shortest paths is twofold. First, it is required so that the approach scales to large software (up to thousands of nodes and edges as shown later in the evaluation). Second, it reflects the idea that at run time, shortest paths are more likely to be executed and propagate the error than longer ones. Moreover, it may be theoretically possible to consider all the paths, nevertheless it is impossible in practice.

2) Prediction: The prediction happens when a developer wants to debug a new fault. The prediction algorithm takes as input a set T_f of failing tests and returns the list of methods

Algorithm 2: Vautrin's prediction algorithm using a causal graph L and failing tests T_f to estimate the suspiciousness of a method.

Input: L a causal graph. T_f the list of failing tests.
Output: I: ranked suspicious nodes

```
1 begin
2     I ← all nodes of L
      /* Compute graph suspiciousness    */
3     for each t ∈ T_f do
4         F ← transitive closure from t in L
5         I ← I ∩ {F}
6     S ← compute spectrum suspiciousness
7     sort elements in I according to S
8     return I
```

ranked according to their suspiciousness. It takes as input the causal graph, first computes graph-suspiciousness, then computes spectrum suspiciousness and combines both into a localization diagnosis. This is given in Algorithm 2.

On line 2, all nodes in L are put into the set I, which means that by default all nodes are considered as graph-suspicious. Then, each failing test of the fault under debug is explored (line 3). For each failing test, we compute the nodes belonging to the transitive closure from the failing test (line 4). The resulting set of nodes is intersected with I (line 5). In this manner, by progressively exploring all failing tests, nodes are removed from the set I, because they are not considered as graph-suspicious.

Finally, a standard spectrum-based fault localization algorithm is used to determine the score of each element in I (line 6). This fault localization algorithm can be any spectrum-based fault localization algorithm. According to our investigations, the best fault localization algorithm to use during prediction phase is the Steimann one [5] (*cf.* Research Question 5); thus, this is the one that is used by default in Vautrin and that we consider in the evaluation. On line 7, elements are ordered in descending order according to the spectrum suspiciousness. The first element is the most suspicious. For instance, considering again Figure 1(b), one can assume that a good spectrum-based suspiciousness score ranks m_3 as more suspicious than m_4.

Note that I may be empty if the causal graph is disconnected or incorrect (that is, the approximation of the causality of some edges is wrong). In such a scenario, Vautrin is not able to predict anything based on graph suspiciousness. Thus, it uses a *fallback mode* in which the scores are computed without using the causal graph, only using spectrum suspiciousness. Consequently, Vautrin necessarily gives results that are at least as good as the one returned by the underlying spectrum fault localization algorithm. In other words, Vautrin never degrades the spectrum-based diagnostic if it is already good.

III. Evaluation

We present the evaluation of Vautrin based on the following research questions.

Research Question 1 Does Vautrin localize faults with less wasted effort than the state of the art?

Research Question 2 Does Vautrin give better perfect predictions than the state of the art?

Research Question 3 What is the execution time cost of fault localization with Vautrin?

Research Question 4 To what extent does Vautrin fall back to the traditional, graph-less, fault localization?

Research Question 5 What is the best spectrum metric to be used with Vautrin?

A. Evaluation Protocol

To evaluate Vautrin, we measure its ability to localize the source of a fault. To that end, we use a dataset that has been proposed by Steimann et al. at ISSTA'13 [5], and the empirical results published therein.

This dataset is based on a set of 10 subject programs. For each software, a set of mutants has been generated. For each mutant, a set of tests is executed; some pass, others fail leading to a "fault". Then, the question is: which method is at the origins of the fault? The performance measured by a fault localization method depends on the set of mutants being derived from software, and from the set of tests being executed. Measuring this performance on a single set of mutants and test nodes is not meaningful as it is not reflecting the performance of the fault localization method in general, that is for any set of mutants and for any set of tests. This is a usual issue faced in machine learning which may be addressed by performing a cross-validation. In our context, this consists in partitioning at random the set of data into 10 subsets and repeatedly training the method with all but one subset, and then measuring its performance on the 10th; finally, the overall performance of the method is the mean of these 10 measurements.

For the sake of open science and reproducible research, our code and experimental data are publicly available on Github[2].

1) Evaluation Metrics: In this section, we present the metrics we use to evaluate our approach. We use the *wasted effort* to determine the accuracy of a fault localization algorithm. For each fault, the fault localization algorithm assigns a suspiciousness score to each method. Let us denote m^* the method at the origins of the fault. Then for this fault, the wasted effort is simply the number of methods that will be investigated before m^* is investigated; phrased in other terms, the wasted effort is the number of methods which score is larger that the score of m^*.

[2]https://github.com/v-m/PropagationAnalysis (causal graph inference and evaluation framework), https://github.com/v-m/PropagationAnalysis-dataset (dataset)

Program	Version	LOC	#Classes	#Tests	#Mutants	Graph	
						#Nodes	#Edges
AC Codec	1.3	2,446	25	188	543	488	825
Daikon	4.6.4	147,153	1,109	157	352	48,689	63,250
Draw2d	3.4.2	22,895	317	89	570	12,298	16,091
Eventbus	1.4	3,572	53	91	577	2,377	2,930
Htmlparser	1.6	21,764	161	600	599	7,905	11,895
Jaxen	1.1.5	12,466	205	695	600	3,171	6,513
Jester	1.37b	1,621	46	64	411	467	645
Jexel	1.0.0b13	1,349	46	335	537	984	1,753
Jparsec	2.0	4,950	122	510	598	5,263	6,723
AC Lang	3.0	18,400	135	1,666	599	6,730	8,906
Total		**236,616**	**2,219**	**4,395**	**5,386**	**88,372**	**119,531**

TABLE I
DESCRIPTIVE STATISTICS OF THE FAULT DATASET USED IN THIS EXPERIMENT. # OF NODES AND EDGES RESPECTIVELY CORRESPOND TO THE NUMBER OF METHODS AND THE NUMBER OF METHOD CALLS.

Equation (1) defines the wasted effort in a more formal way. M is the number of methods being considered and $S(m^*)$ is the score for method m^*.

$$W(m^*) = \sum_{i=1}^{M} 1[S(i) < S(m^*)] + \frac{\sum_{i=1}^{M} 1[S(i) = S(m^*)]}{2} \tag{1}$$

The wasted effort estimates the effort for a developer to find the origin of a fault if he/she considers all suspicious methods ordered by their suspiciousness score. However, we think this metric, even if valuable, is not representative of the ability of a fault localization algorithm to assist fault localization. When a developer uses a tool to assist him/her to localize a fault, he/she wants to save time: if he/she gets a list of suspicious methods, he/she will probably have a look at the first one, but if this method is not the faulty one, he/she will hardly have a look to the following method(s).

This is the reason why we use a second evaluation metric we call the *perfect prediction*. This metric is the number of faults for which the wasted effort is zero. In other words, this metric is the number of faults for which the fault localization algorithm proposes the faulty method (m^*) at the top of the list of suspicious elements, at position #1. It corresponds to the evaluation metric "is in top-k" (used e.g. in [11]) with $k = 1$. Formally, the perfect prediction P is given in Equation (2) with N being the number of faults being considered. An other evaluation metric is MAP (mean average precision), but it does not reflect as well as *perfect prediction* the fact that the developer builds trust based on the top result.

$$P = \sum_{f=1}^{N} 1[W_f = 0] \tag{2}$$

2) Comparison: We conduct a comparative evaluation. We consider 5 fault localization algorithms of the literature: Tarantula[12], Ochiai[6], Zoltar[13], Naish[7] and Steimann[5]. Tarantula and Ochiai are largely used in the fault localization literature, *e.g.* [6], [7], [14]. Thus, for historical reasons, we consider those fault localization algorithms, even if they are not among those performing the best. Zoltar, Naish and Steimann are currently the most accurate fault localization algorithm. Shu et al. [2] also do method-level fault localization. We considered them for a quantitative comparison, however, their heavyweight implementation is not available (we have asked for it). Consequently, we only perform a qualitative comparison in the related work section.

In the remaining of this section, we present these fault localization algorithms. When running the software tests T, we obtain: T_p the set of passing tests and T_f the set of failing tests. If we consider an execution from the point of view of a specific code element e, only a subset of T does actually call the code element e. Let E be the set of tests which actually call e. Then, E_p is a subset of T_p made only of passing tests actually calling e. E_f is a subset of T_f made only of failing tests actually calling e. N is the set of tests which do not call e, that is $N = T - E$, and accordingly $N_p = T_p - E_p$ and $N_f = T_f - E_f$.

According to these sets, a *fault localization algorithm* is used to assign a *suspiciousness score* to each code element e. The higher the score, the higher the chance the code element is faulty. As this suspiciousness is determined using the spectrum-based fault localization, we refer to it in this paper as the *spectrum suspiciousness*. In this paper, we consider the following suspiciousness metrics: Tarantula which has been proposed by Jones et al. in 2002 [12]; Ochiai, a biology metric also used as fault localization algorithm by Abreu et al. [6]; Zoltar which has been proposed by Gonzalez in 2007 [13]; Naish et al. proposed a fault localization algorithm that is perfect under certain assumptions [7]; and Steimann et al.'s metric $T*$ proposed in 2013 [5].

3) Dataset: We consider the dataset proposed by Steimann et al. in ISSTA '13 [5]. Their dataset is made of 10 subject programs (they call them "probands") totalling 5386 one-point mutants. The dataset is composed of execution information for the non-mutated subject program (*i.e.* the reference) and a collection of mutated version of the subject program. It provides information for each mutated version of the program: (i) the list of executed tests, (ii) their execution result (passing or failing), (iii) the list of methods contained in the program, (iv) the mutation operator if applicable and (v) a list of

Software	Tarantula 2002	Ochiai 2006	Zoltar 2007	Naish 2011	Steimann 2013	Vautrin 2016	Improvement
AC Codec	6.01	3.08	2.85	2.86	2.66	**1.81**	47%
Daikon	142.07	125.22	124.78	149.30	124.65	**121.07**	3%
Draw2d	35.41	25.26	23.98	34.12	24.01	**15.46**	55%
Eventbus	17.56	6.16	9.69	50.51	5.99	**4.42**	36%
Htmlparser	21.59	6.11	5.13	21.20	4.82	**3.30**	46%
Jaxen	49.62	18.29	9.27	12.00	11.30	**7.59**	49%
Jester	4.60	2.76	2.55	2.34	2.38	**1.57**	52%
Jexel	15.96	9.06	7.06	6.69	6.64	**5.65**	18%
Jparsec	15.62	3.95	**3.00**	21.90	4.39	3.53	24%
AC Lang	4.87	2.76	2.69	18.10	2.68	**2.40**	12%
Average	**31.33**	**20.27**	**19.10**	**31.90**	**18.95**	**16.68**	**14%**

TABLE II
AVERAGE WASTED EFFORT, IN NUMBER OF INSPECTED METHODS, FOR DIFFERENT FAULT LOCALIZATION ALGORITHMS OVER THE BENCHMARK OF [5]. THE LOWER, THE BETTER. THE BEST SCORES ARE BOLD-FACED.

methods executed by each test (the method-level spectrum).

We produce the subject program call graph using the *soft-miner* tool of our framework. The dataset totals 88,372 nodes and 119,531 edges. Table I shows the 10 subject programs under study. The first column is the subject program name, the second is the considered version, the third is the number of lines of code[3], the fourth is the number of classes for the subject program, the #Tests column contains the number of tests in the program, and the #Mutants gives the number of available one-change mutants (*i.e.* a mutant where only one point in the code has been changed). The last two columns are the call graph information: the number of nodes and the number of edges which correspond respectively to the number of methods and the number of method calls. All the programs are daily used ones and consist in a total of 230,000+ lines of code and 4,000+ test cases. They can be considered as realistic.

B. Empirical Results

Research Question 1 Does Vautrin localize faults with less wasted effort than the state of the art?

As presented in Section III-A1, the wasted effort is the number of wrongly predicted methods returned by the fault localization algorithm before finding the good one. The lower this value, the better, because it directly relates to the time spent by a developer to analyze inaccurate results. For instance, if the wasted effort is 5, this means that a fault localization algorithm has reported 5 non-faulty methods before reporting the actually faulty one.

Table II shows the average wasted effort for each subject program of our dataset[4]. The first column is the program name and the second to the sixth column give the average wasted effort for respectively Tarantula, Ochiai, Zoltar, Naish and Steimann (the unit is an absolute number of methods to inspect, because in this paper, we perform fault localization at the method level). The two last columns are the average wasted effort with our fault localization algorithm, Vautrin and the relative improvement obtained using Vautrin compared to Steimann.

As an example, if we consider the Jaxen subject program, the faulty method is ranked on average at the 11^{th} position using Steimann's fault localization algorithm. For the same subject program, it is ranked at the 7^{th} position with Vautrin. From a software engineering point-of-view, this means that a developer who uses our fault localization algorithm will have to analyze 3.71 less methods on average if he uses Vautrin's fault localization algorithm instead of the Steimann's one.

First, by comparing existing fault localization algorithm listed in Table II (column 2-6), on this dataset, the best fault localization algorithm so far is Steimann: it scores better in 6 cases out of 10. Thus, in the rest of this question, we always refer to it for comparison, and consider it at "the state-of-the-art" (we note that this is always qualified with respect to the dataset under consideration, another system may be better on another dataset). In all cases, Vautrin improves the performances of Steimann's fault localization algorithm as the wasted effort values are always lower for Vautrin. The lowest relative improvement is for Daikon which have a wasted effort of 124.65 methods using Steimann and 121.07 methods using Vautrin, which represents a relative improvement of only 3%. The highest relative and absolute improvement is for Draw2d. Its relative improvement is 55%, with a wasted effort of 24.01 using Steimann and 15.46 using Vautrin. This represents an absolute improvement of 8.55 methods. In other words, a developer would not waste his time in uselessly inspecting 8 methods. The only case on which Vautrin has worse results than one of the other considered fault localization algorithms is for Jparsec for which the wasted effort goes from 3 using Zoltar to 3.53 using Vautrin.

Recall that our core intuition is that the spectrum-based fault localization algorithm misses causal information about the propagation of a fault in the program. Using our approach based on call graph enriches the fault localization process with causal information. Our empirical observations validate this core intuition. Using a causal graph inferred from the call graph for filtering out suspicious methods gives better results.

[3]computed using CLOC (http://cloc.sourceforge.net/)

[4]the lower the better in general, there may be isolated bugs that are better localized with one approach even if the average performance over all bugs is worse

Software	#Faults	Tarantula 2002	Ochiai 2006	Zoltar 2007	Naish 2011	Steimann 2013	Vautrin 2016	Improvement
AC Codec	543	77	221	228	237	237	**251**	6%
Daikon	352	27	38	38	39	39	**42**	8%
Draw2d	570	48	84	85	87	86	**106**	23%
Eventbus	577	26	131	135	103	148	**163**	10%
Htmlparser	599	113	193	197	200	204	**237**	16%
Jaxen	600	121	229	252	257	249	**301**	21%
Jester	411	30	45	45	49	49	**105**	114%
Jexel	537	86	293	301	341	331	**349**	5%
Jparsec	598	69	199	206	216	214	**325**	52%
AC Lang	599	233	384	392	407	408	**431**	6%
Total	**5,386**	**830**	**1,817**	**1,879**	**1,936**	**1,965**	**2,310**	**18%**

TABLE III
NUMBER OF PERFECT PREDICTIONS FOR DIFFERENT FAULT LOCALIZATION ALGORITHMS (I.E. THE FAULTY METHOD IS RANKED AT TOP). THE HIGHER, THE BETTER. THE BEST SCORES ARE BOLD-FACED.

ANSWER TO RESEARCH QUESTION 1
On the considered dataset, Vautrin consistently improves the wasted effort for method-level fault localization, from 3% to 55%, with an average of 14%.

Research Question 2 Does Vautrin give better perfect predictions than the state of the art?

As presented in Section III-A1, a perfect prediction is a prediction where the faulty method is ranked at the top, and is the single method predicted at rank #1 (*i.e.* has a wasted effort equal to zero). In such a case, the developer does not wait a single minute, and the method that he starts to analyze is the one in which he will write the fix.

Table III reports the number of perfect predictions for the faults in the dataset (on average of the cross-validation). The first column is the subject program name, the second column is the number of fault considered and the third to the seventh column give the perfect prediction rate for respectively Tarantula, Ochiai, Zoltar, Naish and Steimann. The two last columns are the number of perfect predictions with our fault localization algorithm, Vautrin and the relative improvement obtained using Vautrin compared to Steimann.

As an example, if we consider the 598 faults for JParsec subject program in the dataset, there are 214/598 of them (36%) for which Steimann makes a perfect prediction. For the same subject program, Vautrin's perfect predictions are 325/598 cases (54%), which represents a relative improvement of 52%.

If we observe only the fault localization algorithm under comparison (and not our technique), we observe that Naish and Steimann are the two fault localization algorithm with the highest number of perfect predictions. In 4 cases out of 10 Naish gives the highest number of perfect predictions, in 3 cases out of 10, Steimann does, and in the 3 remaining cases, both have the same number of perfect predictions. Now, we compare against Steimann as we have done in Research Question 1. Thus, the improvement may be slightly overestimated in cases where Naish reports better perfect prediction scores than Steimann.

For 10 programs out of 10, Vautrin obtains a higher number of perfect predictions than using Steimann. The best relative improvement is for Jester for which the number of perfect predictions is 49 for Steimann and 105 for Vautrin, which represents a relative improvement of 114% (more than twice as many perfect perfections). The smallest relative improvement is for Jexel which goes from 331 with Steimann to 349 with Vautrin, that is, an improvement of 5%.

ANSWER TO RESEARCH QUESTION 2
To sum up, Vautrin also achieves the best result according to the amount of perfect predictions. On the benchmark under consideration, there are 18% more faults which are perfectly predicted using our technique.

Research Question 3 What is the execution time cost of fault localization with Vautrin?

As presented in Section III-A, our approach is composed of four steps: generating the graph, performing mutation analysis, inferring the causal graph and predicting faulty elements when facing a fault. Since we use an existing dataset, we do not have two important measures. The first is the time needed for the generation of mutants. The second is the time required for analyzing the subject program spectrum, *i.e.* the execution of tests for obtaining the propagation paths. Those are used to compute spectrum suspiciousness.

We compute the time cost for all steps but the two cited and report them in Table IV. The first column is the name of the subject program. The second column is the time for generating the call graph. The third column is the time required for computing causal edges based on mutation results. This time is for one fold in our setup, *i.e.* for 90% of the available mutants (*e.g.* 488 mutants for codec). The last column is the average time for predicting the faulty method for one single fault. All times are expressed in seconds. All experiments were made on a HP EliteBook 8570w Mobile Workstation, i7-3740QM quad core, 2.7Ghz, under Arch Linux. As an example, let us consider Jester: each of the three steps lasts less than one

	Offline		Online
	Graph	Causal Inference	Graph-Susp.
AC Codec	1s	$<$ 1s	$<$ 1s
Daikon	27s	$<$ 1s	$<$ 1s
Draw2d	2s	2s	$<$ 1s
Eventbus	2s	$<$ 1s	$<$ 1s
Htmlparser	2s	6s	$<$ 1s
Jaxen	2s	8s	$<$ 1s
Jester	$<$ 1s	$<$ 1s	$<$ 1s
Jexel	$<$ 1s	$<$ 1s	$<$ 1s
Jparsec	2s	$<$ 1s	$<$ 1s
AC Lang	4s	$<$ 1s	$<$ 1s
Average	**4s**	**2s**	$<$ **1s**

TABLE IV
TIMES (IN SECONDS) REQUIRED FOR EACH STEP OF VAUTRIN FOR WHICH WE HAVE THE MEASURES. THE MUTATION AND SPECTRUM ANALYSIS TIME IS NOT REPORTED IN THE BENCHMARK PAPER [5].

second.

If we take a look at the graph generation times, we observe that the generation of 9 out of 10 graphs takes less than 5 seconds. The only exception is Daikon with the slowest time: 27 seconds. The average time is 4 seconds. The graph inference time is generally fast as it takes less than a second in 7 cases out of 10. In the three remaining cases, it takes up to 8 seconds. These two phases are meant to be done offline, for instance every night on a continuous integration server. This experiment suggests that the graph building phase and the graph inference phase do not take too long for this scenario. However, we expect the time for mutation analysis to be much larger.

Regarding the prediction times, it always takes less than 1 second (with an average time of 45ms). This step is meant to be done on-the-fly within the development environment. To this extent, it is acceptable for developers to wait for a couple of milliseconds to get the fault localization diagnosis.

Considering that the main limitation is the time required for generating and executing mutants, one may speed up the mutation generation process using an alternative approach such as one presented by Zhang et al. [15].

ANSWER TO RESEARCH QUESTION 3
For developer usage, Vautrin does not impose a significant overhead compared to spectrum-based fault localization. In addition to the time required to run the test suite, it adds a step which lasts less than 1 second.

Research Question 4 To what extent does Vautrin fall back to the traditional, graph-less, fault localization?

As presented in Section II-C2, Vautrin uses a causal graph to filter out suspicious methods. However, it happens that the intersection of reachable nodes is empty. In this case, Vautrin returns fallback ranking from the Steimann fault localization algorithm. We analyze the number of cases with fallback from the results already discussed in Research Question 1 and 2. For the sake of space, we do not report the whole data.

The worst case is Daikon, for which Vautrin falls back in 82% of the time which means that for 290 faults over 352, we are not able to improve the score given by Steimann fault localization algorithm. For Daikon, Vautrin is able to perform graph-based causal reasoning in 62 cases. On the other side, for 5 subject programs out of 10, fallback happens in less than 25% of the considered faults: Jester, Jexel, Codec, Htmlparser and Eventbus which fall back in respectively 8%, 12%, 15%, 22% and 22% of the faults. For Jester, Vautrin does graph-based reasoning in 378 faults over 411. In total, for 3,883 faults over 5,386, Vautrin predicts faulty elements based on the intersection of transitively reachable nodes.

ANSWER TO RESEARCH QUESTION 4
For the majority of faults (72%), Vautrin has enough information to go beyond simple spectrum-based analysis and to perform graph-based reasoning.

Research Question 5 What is the best spectrum metric to be used with Vautrin?

As presented in Section II-B, within an equivalence class of graph-suspicious elements, Vautrin uses a spectrum-based metric to assign scores to rank suspicious elements. In all experiments, we have used Steimann for spectrum suspiciousness, because it is the best according to the experiment reported in Table II. What if we use Vautrin with the other fault localization algorithm presented in Section III-A2? We now report on the fault localization effectiveness with other spectrum suspiciousness plugged into Vautrin. We note *Vautrin/Y*when we consider Vautrin using the score obtained using the Y fault localization algorithm. Thus, Vautrin/Steimann stands for our approach using the scores obtained using the Steimann fault localization algorithm. Due to space limitation, we do not report the whole data.

We observe that Vautrin/Steimann is the best combination in 4 cases out of 10 for the average wasted effort and 7 cases out of 10 for the perfect prediction. Vautrin/Zoltar is the best in 5 cases out of 10 for the average wasted effort and 1 cases out of 10 for the perfect prediction. Vautrin/Naish is the best in 1 case out of 10 for the average wasted effort and in 6 cases out of 10 for the perfect prediction. Regarding wasted effort, Vautrin/Zoltar may be a better alternative (5 versus 4 for Vautrin/Steimann). But, we have to keep in mind that this combination always produces worse results when we consider perfect predictions. Regarding perfect prediction, Vautrin/Naish is an acceptable alternative (6 versus 7 for Vautrin/Steimann), yet not good for wasted effort. This observation suggests that those two evaluation metrics are not necessarily completely correlated: they capture two different aspects of the fault localization process. If we want to maximize effectiveness with respect to both evaluation metrics (wasted effort and perfect prediction), the best candidate seems to be the Vautrin/Steimann fault localization algorithm, which further validates the choice of Steimann's suspiciousness met-

ric as default choice.

In addition, we setup a small experiment which consists in using a random function for computing the suspiciousness score. Naturally, this experiment shows that the wasted efforts with such a random spectrum fault localization algorithm are really bad (ranging from 74 to 1,058). But, we also observe that applying the graph fault localization algorithm on top of random scores, gives a minimum improvement of 19%, an average improvement of 211% and a maximum improvement up to 540%. This shows that the causality approximation by computing causal edges is indeed effective, even using the worst possible suspiciousness metric one can imagine.

ANSWER TO RESEARCH QUESTION 5
To sum up, to discriminate within an equivalence class of graph-suspicious elements, the best spectrum-based metric to be used with Vautrin is Steimann's as it outperforms the other ones as much with respect to the wasted effort as with respect to the number of perfect predictions.

IV. THREATS TO VALIDITY

Our results are of computational nature. A major bug in our software can invalidate our findings. We have published all our code on Github so as to facilitate reproduction and falsification of our results, if necessary.

Regarding causal graph inference, the quantity and the characteristics of the mutants can impact our findings. The characteristics of the mutation include the operator and the candidate element for mutation. It may be possible that the considered operators and elements may not be the best ones for approximating the causality. However, to avoid being biased by the dataset, we considered an external, peer-reviewed one, that is unbiased with respect to our approach. Also, the dataset may contain equivalent mutants leading to poor coverage or, worse, to biases in the results due to "artificial inflation" phenomenon [16].

We consider in this paper the 10 Java subject programs from Steimann et al.'s dataset to conduct our experiments. However, those 10 programs may have specific graph structures due to developer choices and/or to the used programming language. As a consequence, our results may only be valid for Java subjects, or even worse, only valid for the subject programs under study.

V. RELATED WORKS

Fault localization aims to localize the faulty position in programs. Classical spectrum-based fault localization techniques, already discussed in this paper, include Tarantula by Jones et al. [12] and Ochiai and Jaccard [17]. Xie et al. [18] propose a theoretical analysis on multiple ranking metrics of fault localization and divide these metrics into categories according to their effectiveness. None of them uses graph-based approximation of causality as we do in this paper.

Santelices et al. [14] combine multiple types of code coverage to find out the faulty statements in a program. Baah et al. [19] employ an outcome model to find out the dynamic program dependencies for fault localization. Xu et al. [20] develop a noise-reduction framework for localizing faults. DiGiuseppe and Jones [21] recently propose a semantic fault diagnosis approach, which employs natural language processing to detect the fault locations. Xuan and Monperrus [11] develop a learning-based approach to combine multiple ranking metrics for fault localizing. While they all use novel sources of information for fault localization, they do not outperform the best spectrum-based techniques as we do in this paper.

To improve fault localization, one can also select subsets of tests or modify them. Baudry et al. [22] leverage the concept of dynamic basic blocks to maximize the ability of diagnosing faults with a test suite. Hao et al. [23] propose a test-input reduction approach to reduce the cost of inspecting the test results. Gong et al. [24] design a diversity-maximization-speedup approach to reduce the manual labeling of test cases and improve the accuracy of fault localization. Yoo et al. [25] address the problem of fault localization prioritization. Xuan and Monperrus [26] extract test slices related to failing assertions to improve fault effectiveness. Those techniques are orthogonal to the one presented in this paper. Future work can combine test case selection and graph-based causality to further improve fault localization.

Baah et al. [1] propose to use regression models as a fault localization algorithm. Those models are built using spectrum information and data extracted from a program dependence graph (*i.e.* at the statement granularity). They assess their approach on faults from the Siemens, Sed and Space datasets. Shu et al. [2] use a similar approach but based on dynamic call graphs (*i.e.* at the method granularity) mixed with dynamic data dependencies. They assess their approach on faults randomly selected from a bug database for four programs. Those two papers, as us, consider that fault localization is a causal inference problem in essence. Also, they use graph information (respectively program dependence graphs [1] and dynamic call graphs [2]) as basis for causal reasoning; in our case, we use a different graph: a static class hierarchy analysis (CHA) call graph. The key novelty of our approach is that we use mutation analysis to approximate causal relationships. A quantitative comparison is future work, because their implementation is not publicly available, and having asked for it, is not yet ready for sharing.

A. Mutation and Fault Localization

Mutation-based fault localization has been recently proposed. The kernel idea of mutation-based fault localization is to localize faults by injecting faults. Zhang et al. [27] propose FIFL, a fault injecting approach to localize faulty edits in evolving Java programs. Candidate edits are ranked based on the suspiciousness of mutants. Papadakis and Le Traon [28] develop Metallaxis-FL, a mutation-based technique for fault localization on C programs. Their work shows that test cases that are able to kill mutants can enable accurate fault localization. Moon et al. [29] propose MUSE, an approach based on both mutants of faulty statements and mutants of

correct statements. None of them explores the confluence of mutants and graph analysis as we propose in this paper.

VI. Conclusion

We have presented Vautrin, a novel approach for fault localization. Vautrin is based on the idea of approximating causal effects at the method level: it consists of extracting an approximate causal graph out of a call graph based on execution information obtained from mutation testing. We have evaluated our approach on the dataset by Steimann et al. The evaluation setup results in 5,386 fault localization diagnosis. Overall, Vautrin is able to make 2,310 perfect predictions, which means that it predicts ranked methods on which the top ranked one is indeed the faulty one.

Future work will explore how to speed up the causal graph approximation process by tailored mutation analysis: where to create mutants so as to maximize the quantity of new causal information that is inferred.

References

[1] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal Inference for Statistical Fault Localization," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2010, pp. 73–84.

[2] G. Shu, B. Sun, A. Podgurski, and F. Cao, "MFL: Method-Level Fault Localization with Causal Inference," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2013, pp. 124–133.

[3] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault Localization using Execution Slices and Dataflow Tests," in *Proceedings of the International Symposium on Software Reliability Engineering*, Oct. 1995, pp. 143–151.

[4] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An Empirical Investigation of Program Spectra," in *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, 1998, pp. 83–90.

[5] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-based Fault Locators," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2013, pp. 314–324.

[6] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, 2006, pp. 39–46.

[7] L. Naish, H. J. Lee, and K. Ramamohanarao, "A Model for Spectra-based Software Diagnosis," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[8] T. D. Nielsen and F. V. Jensen, *Bayesian Networks and Decision Graphs*. Springer Science & Business Media, 2009.

[9] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction in Object-oriented Languages," in *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1997, pp. 108–124.

[10] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, "Mutation Analysis," Yale University, Department of Computer Science, Tech. Rep., 1979.

[11] J. Xuan and M. Monperrus, "Learning to Combine Multiple Ranking Metrics for Fault Localization," in *Proceedings of the International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 191–200.

[12] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," in *Proceedings of the International Conference on Software Engineering*, 2002, pp. 467–477.

[13] A. Gonzalez, "Automatic Error Detection Techniques based on Dynamic Invariants," Ph.D. dissertation, MS thesis, Delft University of Technology, The Netherlands, 2007.

[14] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight Fault-localization Using Multiple Coverage Types," in *Proceedings of the International Conference on Software Engineering*, 2009, pp. 56–66.

[15] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive Mutation Testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2016, pp. 342–353.

[16] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Traon, "Threats to the Validity of Mutation-based Test Assessment," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2016, pp. 354–365.

[17] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 89–98.

[18] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 4, pp. 31:1–31:40, Oct. 2013.

[19] G. K. Baah, A. Podgurski, and M. J. Harrold, "Mitigating the Confounding Effects of Program Dependences for Effective Fault Localization," in *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering*, 2011, pp. 146–156.

[20] J. Xu, Z. Zhang, W. K. Chan, T. H. Tse, and S. Li, "A general noise-reduction framework for fault localization of Java programs," *Information and Software Technology*, vol. 55, no. 5, pp. 880–896, May 2013.

[21] N. DiGiuseppe and J. A. Jones, "Semantic Fault Diagnosis: Automatic Natural-language Fault Descriptions," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2012, pp. 23:1–23:4.

[22] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving Test Suites for Efficient Fault Localization," in *Proceedings of the International Conference on Software Engineering*, 2006, pp. 82–91.

[23] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei, "Test input reduction for result inspection to facilitate fault localization," *Automated Software Engineering*, vol. 17, no. 1, pp. 5–31, Aug. 2009.

[24] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity Maximization Speedup for Fault Localization," in *Proceedings of the International Conference on Automated Software Engineering*, 2012, pp. 30–39.

[25] S. Yoo, M. Harman, and D. Clark, "Fault Localization Prioritization: Comparing Information-theoretic and Coverage-based Approaches," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 3, pp. 19:1–19:29, Jul. 2013.

[26] J. Xuan and M. Monperrus, "Test Case Purification for Improving Fault Localization," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2014.

[27] L. Zhang, L. Zhang, and S. Khurshid, "Injecting Mechanical Faults to Localize Developer Faults for Evolving Software," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications*, 2013, pp. 765–784.

[28] M. Papadakis and Y. Le Traon, "Metallaxis-FL: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, Aug. 2015.

[29] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the Mutants: Mutating Faulty Programs for Fault Localization," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2014, pp. 153–162.