2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Tracking Static Analysis Violations Over Time to Capture Developer Characteristics

Pavel Avgustinov, Arthur I. Baars, Anders S. Henriksen, Greg Lavender, Galen Menzel,
Oege de Moor, Max Schäfer, Julian Tibble
Semmle Ltd.
Oxford, United Kingdom
publications@semmle.com

***Abstract*—Many interesting questions about the software quality of a code base can only be answered adequately if fine-grained information about the evolution of quality metrics over time and the contributions of individual developers is known. We present an approach for tracking static analysis violations (which are often indicative of defects) over the revision history of a program, and for precisely attributing the introduction and elimination of these violations to individual developers. As one application, we demonstrate how this information can be used to compute "fingerprints" of developers that reflect which kinds of violations they tend to introduce or to fix. We have performed an experimental study on several large open-source projects written in different languages, providing evidence that these fingerprints are well-defined and capture characteristic information about the coding habits of individual developers.**

I. Introduction

Static code analysis has become an integral part of the modern software developer's toolbox for assessing and maintaining software quality. There is a wide variety of static analysis tools, particularly for Java and C/C++, which examine code for potential bugs or performance bottlenecks, flag violations of best practices, and compute software metrics. It has been shown that static analysis warnings and sub-par metric scores (below collectively referred to as *violations*) are indicative of software defects [27], [29], [35], [37], [38].

Some static analysis tools are tightly integrated into the development cycle and examine code as it is being written [1], [6], while others run in batch mode to produce detailed reports about an entire code base [4], [7], [31]. The latter approach is particularly useful for obtaining a high-level overview of software quality. Moreover, most modern tools can aggregate violations at different levels of detail, thus making it easy, for instance, to identify modules that appear to be of worse quality than others and should thus receive special attention.

However, this view of software quality is both *static* and *coarse-grained*: it is static because it only concerns a single snapshot of the code base at one point in time, and it is coarse-grained because it does not differentiate contributions by individual developers.

Most software is in a constant state of flux where developers implement new features, fix bugs, clean up and refactor code, add tests, or write documentation. While existing tools can certainly analyse multiple versions of a code base separately, they cannot easily assess changes between revisions. For instance, we might want to know what violations were fixed in a given revision, or which new violations were introduced. In short, analysing individual snapshots cannot provide an accurate picture of the evolution of software quality over time.

Similarly, most code bases have many authors: there may be seasoned developers and novice programmers, prolific core hackers and occasional bugfix contributors. By just looking at a single snapshot, it is impossible to understand the quantity and quality of contributions of individual developers.

Yet there are many situations where more dynamic and fine-grained information is desirable. We briefly outline three example scenarios:

1) Alice wants to delegate a crucial subtask of her program to a third-party library and is looking for an open-source library that fits her needs. As is usually the case with open-source projects, many different implementations are available, and Alice has to carefully choose one that is not only of high quality but also actively maintained. She could use an off-the-shelf static analysis tool to gain insight into the quality of the latest version of the library. In addition, however, she might want to know how its quality has evolved over time, and how different developers have contributed to the code. For instance, she may want to avoid a library where a core developer who has contributed a lot of new code and bug fixes for many years has recently become inactive.
2) Bob is managing a team of developers, and would like to better understand their strengths and weaknesses. If he knew, for instance, that developer Dave often introduces static analysis violations related to concurrency, he could arrange for Dave to receive additional training in concurrency. If there is another developer on the team who often fixes such violations, Bob could team them up for code reviews.
3) In an effort to improve software quality, Bob's colleague Carol uses a static analyser to find modules that have a particularly high violation density and hence are likely to be of bad quality. In deciding which developer to put to work on this code, it again helps if she understands their expertise. If, for instance, the code in question has many violations related to possible null pointer exceptions, it may be a good idea to assign a developer who has a strong track record of fixing such violations.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.62

All three scenarios rely on being able to precisely attribute new and fixed violations to individual developers. This can be achieved by integrating static analysis with revision control information: if a violation appears or disappears in a revision authored by developer D, then this suggests that D was responsible for introducing or fixing this violation, and it can justifiably be attributed to them.[1]

For this method to work, however, we need a reliable way of tracking violations between revisions: if both revision n and revision $n+1$ exhibit a violation of the same type, we need to determine whether this is, in fact, the same violation, or whether the violation in n was fixed and a new one of the same type just happened to be introduced in $n+1$.

Further challenges to be handled include merge commits (which have more than one parent revision) and un-analysable revisions: in almost any real-world code base, there is bound to be an occasional bad commit that is not compilable, or cannot meaningfully be analysed for other reasons. Such commits require special care in order not to wrongly attribute violations to authors of later commits.

Finally, there is an implicit assumption behind the scenarios outlined above, namely that individual developers have a distinctive "fingerprint" of violations that they introduce or fix. If all developers tend, on average, to make the same mistakes, then attribution information would not be very useful.

In this paper, we present Team Insight, a tool for fine-grained tracking of software quality over time. At its core is a method for tracking violations across revisions based on a combination of diff-based location matching and hash-based context matching. This approach is robust in the face of unrelated code changes or code moves, and fast enough to work on large, real-world code bases. Team Insight integrates with many popular revision control systems to attribute violations to developers. It uses a simple distributed approach for efficiently analysing and attributing a large number of revisions in parallel.

We also present an approach for computing fingerprints from attribution data, which compactly represent a summary of which violations a developer tends to introduce or fix.

We have used Team Insight to analyse the complete revision history of several large open source projects written in Java, C++, Scala and JavaScript. We performed experiments to verify that our violation tracking algorithm does not spuriously match up unrelated revisions, and to gauge the relative importance of the different matching strategies. Finally, we used the information about new and introduced violations to test the robustness of our developer fingerprints: selecting a training set and a test set from the analysed snapshots, we computed fingerprints from both sets of snapshots and compared them. We found that fingerprints were both stable (i.e., the fingerprints computed for a single developer from the test and the training set are very similar) and characteristic of individual developers (i.e., the fingerprints computed for different developers are quite different). Hence our fingerprints could be useful in scenarios such as the ones outlined above.

[1]Many version control systems provide a way to determine which developer last changed a given line of code. This information, however, is not usually enough to determine who introduced a violation: the last developer to touch the relevant source code may have been making an entirely unrelated edit. Also, many violations are non-local in that the code that causes them is far removed from the code that is flagged by the static analysis. Finally, this approach does not provide any information about fixed violations.

```
127 Set<String> revs;
128 ...
162 for (IRevision rev : new ArrayList<IRevision>(keep)) {
163   if (!revs.contains(rev)) {
164     ...
179   }
180 }
```

Fig. 1. Example of a violation

We now turn to a brief exposition of some basic concepts underlying Team Insight (Section II). We then explain our violation matching technique in more detail (Section III) and discuss how it is used to attribute new and fixed violations to developers (Section IV). Next, we describe our approach for computing developer fingerprints in Section V. These developments are then pulled together in Section VI, which reports on our experimental evaluation. Finally, Section VII discusses related work, and Section VIII concludes.

II. Background

We start by motivating the concept of violation matching with a real-world example from a Java project we analysed.

At one point, the file `DeleteSnapshots.java` in the code base contained the code fragment shown in Figure 1. On line 127, the variable `revs` is declared to be of type `Set<String>`, yet the `contains` test on line 163 checks whether `rev`, which is of type `IRevision`, is contained in it.

This is highly suspicious: unless unsafe casts were used to deliberately break type safety, every element in `revs` must be of type `String`, which `rev` is not. Thus, the test must always return **`false`**, which is probably not what the programmer intended. In this case the code should have checked whether `rev.getId()` (which is a string representation of `rev`) is contained in `revs`.

Arguably, this code should be rejected by the type checker, but for historical reasons the `contains` method in Java's collections framework is declared to have parameter type `Object`, so any object can be tested for membership in any collection, regardless of the collection's type. This problem is common enough that many static analysis tools for Java, including our own static analysis tool Project Insight [31], check for it.

To trace the life cycle of this violation, we consider seven revisions of the code base, which we refer to as revisions 0 to 6. Figure 2 gives a brief summary of the relevant changes in each revision, and the source location associated with the violation (here, the call to `contains`). In each case, the violation location is given as a pair of a file name and a line number.[2] For now, we assume that these revisions were committed in sequence as shown in Figure 3 (a).

[2]In practice, of course, locations are more precise: they include a start and an end position, and specify both line and column number.

Revision	Change Summary	Violation Location
0	`DeleteSnapshots.java` created	N/A
1	violation introduced	`DeleteSnapshots.java`:163
2	code added before violation	`DeleteSnapshots.java`:173
3	code added before violation	`DeleteSnapshots.java`:175
4	containing file renamed	`FindObsoleteSnapshots.java`:175
5	code added after violation	`FindObsoleteSnapshots.java`:175
6	violation fixed	N/A

Fig. 2. Relevant revisions of the code in Figure 1

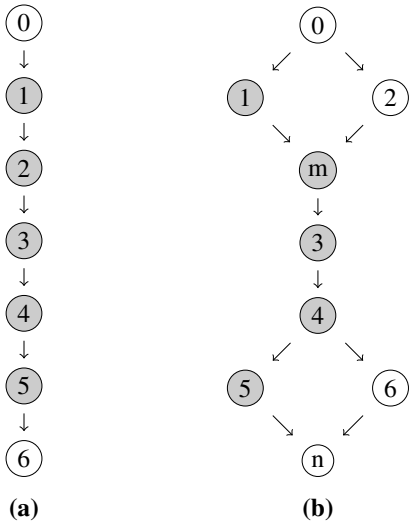

Fig. 3. Two commit graphs for the revisions in Figure 2; revisions where the violation is present are shaded.

The file containing the violation was first created in Revision 0, and the violation itself was introduced in Revision 1; at this point, it was located at line 163 of file `DeleteSnapshots.java`. In Revision 2 the violation was still present, but some code had been inserted before it, so it had moved to line 173. In Revision 3, it moved to line 175. In Revision 4, its line number number did not change, but the enclosing class was renamed to `FindObsoleteSnapshots`, and the enclosing file to `FindObsoleteSnapshots.java`. In Revision 5, the file was again changed, but since the changes were textually after the violation its location did not change. Finally, the violation was fixed in Revision 6.

If we want to automatically and precisely attribute violations to developers, we have to carefully keep track of violations across revisions. For instance, a violation tracking approach based purely on source locations would consider the violation in Revision 2 to be different from the violation in Revision 1, since they have different source locations. Consequently, the author of Revision 2 would erroneously be considered to have fixed a violation on line 163 and introduced a violation of the same type on line 173.

A more lenient location-based approach might try to identify violations based on the name of the method and class enclosing its source location. This, however, would fail in Revision 4, where the enclosing class is renamed (along with the file).

Ignoring source locations entirely, one could attempt to match up violations based on the similarity of their surrounding code: if two subsequent revisions contain two violations of the same type that appear in identical or very similar fragments of code, then there is a good chance that both are occurrences of the same violation. In Revision 4, for instance, there were only two minor textual changes (in addition to the file renaming), so a similarity-based approach could easily determine that the violation is still present and has simply moved to another file.

Note, however, that neither location-based matching nor similarity-based matching is strictly superior to the other: while the former cannot deal with code movement or file renaming, the latter can become confused by unrelated changes in code close to the violation. In such a case, a similarity-based matcher may not be able to identify violations even if their location has not changed.

Team Insight uses a combined approach detailed in the next section: first, it tries to match up as many violations as possible based on their source location. For those violations that could not be matched up, it computes a hash of the surrounding tokens (similar to techniques used in clone detection [17]), and then matches up violations with the same hash.

Additional care has to be taken when considering non-linear commit graphs with branches and merges. Assume, for instance, that the revisions of Figure 2 were committed as shown in the commit graph of Figure 3 (b). Here, Revision 1 and Revision 2 are committed independently on separate branches, and then merged together by a merge commit m. Similarly, Revisions 5 and 6 are committed independently and merged by n. Note in particular that the violation is now no longer present in Revision 2, which branched off before the violation was introduced in Revision 1. In the merge commit m, however, the violation is merged in with the changes from Revision 2.

The author of the merge commit m should clearly not be blamed for introducing the violation, since it is already present in Revision 1. In general, we can only consider a merge commit to introduce a violation if that violation is absent in *all* its parent revisions (but not the merge commit itself).

Similarly, the author of the merge commit n should not be considered to have fixed the violation, since it was already

absent in Revision 5. Again, a merge commit can only be considered to fix a violation if that violation is present in all its parent revisions but not the merge commit itself.

Finally, it should be noted that in any non-trivial code base there are revisions that for one reason or another cannot be analysed. For example, this could be due to a partial or erroneous commits that results in uncompilable code, or simply because some external libraries required by a very old revision are no longer available. Such un-analysable revisions have to be treated with care when attributing violations. If, say, Revision 3 of our example was not analysable, then it would not be possible to decide whether the violation present in Revision 4 was introduced in Revision 4 itself or was already present in Revision 3. Looking further back, we can, of course, note that the revision was already present before Revision 3, so it is likely that neither 3 nor 4 introduced the violation, but this is at best an educated guess.

This concludes our informal discussion. We will now describe the Team Insight attribution algorithm in more detail.

III. Matching Violations

We start by establishing some terminology.

A *project* is a code base that can be subjected to static analysis. A *snapshot* is a version of a project at a particular point in time; for instance, if the project's source code is stored in a version control system, every revision is a snapshot. We assume that there is a parent-of relation between snapshots. There may be multiple snapshots with the same parent (due to branching), and conversely a snapshot may have multiple parents (due to merging).

We do not assume a particular underlying static analysis system. All we require is that the static analysis can, for a given snapshot S, produce a set of *violations*, where each violation is uniquely identified by a source location l and a violation type t. The source location, in turn, is assumed to be given by a start position and an end position delimiting the piece of code that gives rise to the violation. Thus, the violation can be modelled as a triple (S, l, t). We explicitly allow for the case that the static analysis may not be able to analyse some snapshots, in which case there will be no violation triples for that snapshot.[3]

With this terminology in place, the violation matching problem can be stated succinctly as follows:

> Given two snapshots S_p and S_c of the same project, where S_p is a parent snapshot of S_c, and two violation triples $V_p := (S_p, l_p, t_p)$ and $V_c := (S_c, l_c, t_c)$, do V_p and V_c indicate the same underlying defect?

We will not attempt to give a precise semantic definition of when two violations indicate the same defect. Previous studies [21] have shown that even developers familiar with a code base often disagree about the origin of code snippets, so there may not be a single correct definition anyway. Instead, we describe the syntactic violation matching approach taken by Team Insight, and leave it to our experimental evaluation to provide empirical support for its validity.

To decide whether two violations (S_p, l_p, t_p) and (S_c, l_c, t_c) match we employ a combination of three different matching strategies: a location-based strategy that only takes the violation locations l_p and l_c into account, a snippet-based strategy that considers the program text causing the violations, and a hash-based strategy that also considers the program text around the violations.

We will now explore these strategies in detail. Note that clearly two violations can only match if they are of the same type, so in the following we implicitly assume that $t_p = t_c$.

A. Location-based violation matching

The idea of location-based violation matching is to use a diffing algorithm to derive a mapping from source positions in S_p to source positions in S_c, and then match up violations if they have the same or almost the same start position under this mapping.

Our implementation uses the well-known diffing algorithms of Myers [26] and of Hunt and Szymanski [15] to derive source position mappings for individual file in the snapshot. Specifically, we use the former algorithm for dense diffs where there is a lot of overlap between the two files, and the latter for sparse diffs. Clearly, computing pairwise diffs for all the files in the parent and child snapshots would be much too expensive; hence, we only compute diffs for files with the same path in both snapshots. In particular, if a file is renamed or moved to a different directory, the location-based violation matching will not be able to match up any violations occurring in it. We rely on the hash-based matching explained below to catch such cases.

We will use the Java code snippets shown in Figure 4 as our running example in this section. The code on the left, which we assume to be from the parent snapshot, contains three violations: V_1 is an instance of the `contains` type mismatch problem mentioned in Section II; V_2 flags a reader object that (we assume for this example) is not closed, thus potentially leading to a resource leak; V_3 flags a call to `System.gc`, which is generally bad practice. These three violations reappear in the child snapshot on the right, but the statements containing them have been rearranged and new code has been inserted. We will now show how our matching strategies match up these violations.

When given two files F_p and F_c, where F_p is from the parent snapshot and F_c from the child snapshot, the diffing algorithms essentially partition F_p and F_c into sequences of line ranges $r^p_1, \ldots, r^p_n$, and $r^c_1, \ldots, r^c_n$, respectively. Each pair of line ranges (r^p_i, r^c_i) is either a *matching pair*, meaning that the ranges are textually the same, or a *diff pair*, meaning that they are not the same. Every line in F_p and F_c belongs to exactly one line range. For a location l, we write $[l]$ to mean the line range its start line belongs to, and $\Delta(l)$ to mean the distance (in lines) from l to the start of its line range.

[3] In practice, the static analysis may be able to analyse parts of a snapshot, but we do not use this partial information since it is very hard to compare between snapshots.

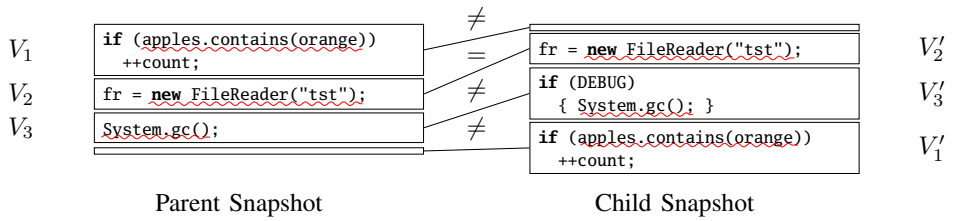

Fig. 4. Examples of location-based violation matching

In general, there is no unique way of performing this partitioning; we simply rely on whatever partitioning the underlying diff algorithm produces.

In our example, there are four line ranges as indicated by the boxes. We draw lines between corresponding line ranges, which are labelled with $=$ for matching pairs and with $\neq$ for diff pairs. $([V_1], [V_1'])$ and $([V_3], [V_3'])$ form diff pairs, while $([V_2], [V_2'])$ is the single matching pair. All violations have a distance of zero from the start of their region, except for V_3' which has $\Delta(V_3') = 1$.

Consider now a violation V_c in the child snapshot at location l_c, and a violation V_p in the parent snapshot at location l_p, where the files containing l_c and l_p have the same path. If their ranges correspond, we try to match them up as follows:

1) If $([l_c], [l_p])$ is a matching pair, then the violations are triggered by code that did not change between S_p and S_c, so we only match them up if their positions within the region are exactly the same, i.e., $\Delta(l_c) = \Delta(l_p)$.
   In our example, there is only one matching pair: the one containing V_2 and V_2', respectively. Since their positions in the region are the same, they are matched up.
2) If, on the other hand, $([l_c], [l_p])$ is a diff pair, we allow the locations to vary slightly: V_c and V_p are considered to match if $|\Delta(l_c) - \Delta(l_p)| \leq \epsilon$ for some threshold value ϵ. In our implementation, we use $\epsilon = 3$.
   For example, consider V_3 on the left and V_3' on the right. Their regions form a diff pair, and $\Delta(V_3) = 0$ whereas $\Delta(V_3') = 1$; since their distance is less than three lines, we match them up.

B. Snippet-based violation matching

Since location-based matching requires violation locations to belong to corresponding line ranges, this matching strategy will fail for violations like V_1 whose location has changed significantly between snapshots. We use an additional strategy to catch simple cases where a violation has moved in the same file, but is triggered by exactly the same snippet of code: if l_c and l_p belong to the same file and the source text of l_c and l_p (i.e., the text between the respective start and end positions) is the same, we match them up. In the example, this allows us to also match up V_1 and V_1'.

C. Hash-based violation matching

Neither location-based matching nor snippet-based matching apply to violations in files that are renamed or moved between snapshots. To match up those violations, we employ a hash-based strategy that tries to match violations based on the similarity of their surrounding code.

Specifically, for a violation $V = (S, l, t)$, we compute two hash values $h_<(V)$ and $h_>(V)$: the former is computed from the n tokens up to and including the first token in l, and the latter is computed from the n tokens starting with the first token in l, where n is a fixed threshold value (by default, we use $n = 100$). Two violations V_c and V_p are considered to match if $h_<(V_c) = h_<(V_p)$ or $h_>(V_c) = h_>(V_p)$.

If the location l starts less than n tokens into the file, $h_<(V)$ is undefined; similarly, $h_>(V)$ is only defined if l starts at least n tokens before the end of the file. This avoids spurious matches due to short token sequences, but it makes it necessary to use two hashes, since otherwise violations near the beginning or the end of a file could never be matched.

IV. Attributing Violations

Now that we have discussed how to match violations between two snapshots, let us consider the problem of *attributing* a snapshot S, that is, computing the sets of new and fixed violations.

Clearly, a snapshot is only attributable if both it and all its parent revisions can be analysed. As a special case, if S does not have any parents, we do not consider it attributable, since it may contain code copied from other sources and we know nothing about the history of the violations in this code.

Let us first consider the case where S has precisely one parent snapshot P. We write $V(S)$ and $V(P)$ for the sets of violations of S and P, respectively. Using the matching strategies described earlier, we can determine for any two violations whether they match.

In general, a single violation in $V(S)$ may match more than one violation in $V(P)$ and vice versa. To decrease the chance of accidentally matching up unrelated violations, we prioritise our matching strategies as follows:

1) Apply location-based matching first. If $v \in V(S)$ is located in a diff range and may match more than one violation in $V(P)$, choose the closest one.
2) Only use snippet-based matching if diff-based matching fails. If more than one violation in $V(P)$ has the same source text as v, choose the closest one.
3) Only use hash-based matching if location-based matching and snippet-based matching both fail. If there are

two or more violations in $V(S)$ or $V(P)$ that have the same hash, exclude them from the matching.[4]

Furthermore, once a violation from $V(P)$ has been matched up with a violation from $V(S)$, we exclude it from further consideration. In this way, we obtain a matching relation $\sim$ that matches every violation in $V(S)$ with at most one violation in $V(P)$ and vice versa.

Now the new violations in S are simply those for which there is no matching violation in P:

$$N(S) := \{v \in V(S) \mid \neg\exists v' \in V(P).v \sim v'\}$$

Dually, the set of fixed violations in S are those violations in P for which there is no matching violation in S:

$$F(S) := \{v' \in V(P) \mid \neg\exists v \in V(S).v \sim v'\}$$

Note that according to this definition, simply deleting a piece of code fixes all the violations in it.

Merge commits have more than one parent. Generalising the definition of $N(S)$ to this case poses no problems, but generalising $F(S)$ is not so straightforward.[5] Thus, we choose not to define any new or fixed violations for merge commits.

Finally, let us consider how to efficiently attribute multiple snapshots. A common use case would be to attribute every snapshot in the entire revision history of a project, or every snapshot within a given time window.

Since all parents of a snapshot need to be analysed before attribution is possible, we could perform a breadth-first traversal of the revision graph: start by analysing the first snapshot and all its children; assuming that all of them are analysable, the child snapshots can then be attributed. Now analyse all of their children for which all parents have been analysed in turn and attribute them, and so on.

In practice, however, later revisions are often more interesting: users normally first want to understand recent changes in code quality before they turn to historic data. Also, older snapshots are more likely to be unanalysable due to missing dependencies, so attribution may not even be possible in many cases. This is why Team Insight attributes snapshots in reverse chronological order.

Since analysing and attributing a large number of revisions can take a very long time, the analysis and attribution tasks need to be parallelised as much as possible. To this end, Team Insight splits up all analysable snapshots into *attribution chunks*: an attribution chunk consists of a set A of snapshots to attribute together with a set U of supporting snapshots such that for each snapshot in A all of its parents are contained in either A or U. Clearly, once all snapshots in $A \cup U$ have been analysed, all the snapshots in A can be attributed without referring to any other snapshots. Thus, different attribution chunks can be processed in parallel.

[4]Note that this case only arises if the violation has disappeared from its original file (since both location-based matching and diff-based matching fail), and multiple new copies of the violation have appeared elsewhere. This is most commonly caused by a piece of code (containing the violation) being cut from its own file and pasted into multiple other files. By avoiding a match, we force this to be considered as a single fixed violation and multiple introduced violations, which seems like the most appropriate way to model it.

[5]$F(S)$ consists of violations in the parent snapshot, and so for merge commits we would have to identify corresponding violations across all parent snapshots. This is made more difficult by the fact that our definition of hash-based matching is not transitive (since only one hash needs to be equal to establish a match), so all pairs of parent snapshots would have to be compared against each other.

V. Developer Fingerprinting

We now discuss an important application of attribution information: computing violation fingerprints to characterise which kinds of violations developers tend to introduce or fix.

We represent the violation fingerprint for a developer by two vectors, one containing information about introduced violations and one about fixed violations. Each vector has one component per violation type, where the component corresponding to some violation type tells us how often the developer introduces or fixes violations of that type.

Given a set of attributed snapshots, we can compute a fingerprint for every developer by simply summing up the number of introduced and fixed violations of every type over all the snapshots authored by the given developer.

However, such uncalibrated fingerprints are difficult to compare between developers: a developer who has changed more lines of code is, in general, expected to have introduced and fixed more violations than a developer with fewer contributions. Thus, it makes sense to scale the components of each vector to account for such differences in productivity.

We consider two scaling factors:

1) Scaling by churn, where the violation counts are divided by the total number of lines of code changed by the developer across the set of considered snapshots.
2) Scaling by total number of violations, where the violation counts are divided by the total number of new/fixed violations of the developer. This provides an overview of what portion of the violations a developer introduces or fixes are of a certain type.

As an example, assume we have two developers d_1 and d_2, and we have a set of snapshots where violations of two types have been attributed. If d_1 has introduced a total of 10 violations of the first type and fixed none, while introducing 5 violations of the second type and fixing 2, her uncalibrated violation fingerprint is $\langle(10, 5), (0, 2)\rangle$. Similarly, if the fingerprint of d_2 is $\langle(4, 1), (0, 4)\rangle$, then this means that he introduced four violations of the first type and one of the second type, while fixing four violations of the second type, but none of the first type.

Given these uncalibrated fingerprints, it may be tempting to deduce that d_1 introduces more violations of the first type than d_2. If, however, d_1 contributed 50000 lines of churn while d_2 only touched 1000 lines of code, this conclusion is unwarranted: d_1 contributed 50 times as much code than d_2, but only introduced about twice as many violations of the first type. Scaling the fingerprints by the amount of churn (in thousands of lines of code) d_1's fingerprint becomes $\langle(0.2, 0.1), (0, 0.04)\rangle$, while d_2's fingerprint is still $\langle(4, 1), (0, 4)\rangle$: this shows that, relatively speaking, d_2 is much

TABLE I
PROJECTS USED IN THE EVALUATION

Name	**Language**	**# Snapshots**	**Size** (KLOC)	**Churn** (KLOC)	**Total New**	**Total Fixed**
Hadoop Common [8]	Java	27086	1200	6206	51294	20212
MongoDB [25]	C++	5057	429	536	2721	2075
Spark [9]	Scala	7226	75	835	8179	3960
Gaia [10]	JavaScript	27024	560	4468	75337	75938

more likely to introduce violations of both types, but is also more likely to fix violations of the second type.

If, instead, we want to compare how many of the fixed/introduced violations of a developer are of a certain type we can scale by the total number of fixed/introduced violations: overall, developer d_1 introduced 15 violations and fixed two, so her fingerprint becomes $\langle(\frac{2}{3}, \frac{1}{3}), (0, 1)\rangle$; developer d_2 introduced five and fixed four, giving the (very similar) fingerprint $\langle(0.8, 0.2), (0, 1)\rangle$.

Which kind of fingerprint is more useful depends on the application area. Fingerprints scaled by churn are useful to compare different developers, and could, for instance, be used to find out which team member is most adept at fixing a given kind of violation. Fingerprints scaled by the number of violations, on the other hand, compare a single developer's performance in different areas, and could hence be used to select appropriate training.

VI. EVALUATION

We now report on an evaluation of our violation matching approach and the developer fingerprinting on four large open-source projects with significant revision histories. For the violation matching, we investigate the quality of the matchings produced, and the relative importance of the different matching strategies. For the fingerprinting, we assess whether fingerprints are, in fact, characteristic of individual developers.

A. Evaluation subjects

As our evaluation subjects, we chose the four open-source projects Hadoop Common, MongoDB, Spark and Gaia, as shown in Table I. For each of our subjects, the table shows the language they are implemented in; the total number of snapshots that were attributed; the approximate size (in thousand lines of code) of the latest attributed snapshot; the total amount of churn across all attributed snapshots; and the total number of new and fixed violations.[6]

To find violations, we used the default analysis suites of our tool Project Insight, comprising 191 analyses for Java, 94 for C++, 115 for Scala, and 73 for JavaScript. The raw analysis results our experiments are based on are available from `http://semmle.com/publications`.

B. Evaluating violation matching

We evaluate our violation matching algorithm with respect to two evaluation criteria:

[6]Note that for Gaia there are more fixed than new violations; this is because some violations were introduced in unattributable revisions.

TABLE II
VIOLATION MATCHINGS CONTRIBUTED BY INDIVIDUAL ALGORITHMS

Project	Exact	Fuzzy	Snippet	Hash
Hadoop	133,000,488	46,880	4,505	14,369
	99.95%	0.04%	0.00%	0.01%
MongoDB	52,432,554	21,551	2,044	21,239
	99.91%	0.04%	0.00%	0.04%
Spark	22,891,706	61,949	1,967	43,379
	99.53%	0.27%	0.01%	0.19%
Gaia	63,855,193	28,830	6,280	39,154
	99.88%	0.05%	0.01%	0.06%

EC1 How many violation matchings are contributed by the different matching algorithms?

EC2 Do these violation matchings match up violations that actually refer to the same underlying defect?

To answer **EC1**, we randomly selected 5000 snapshots from each of our subject programs and counted how many violation matchings were contributed by each of the algorithms. The results are shown in Table II: for the location-based violation matching, we distinguish between exact matches (column "Exact") and matches in diff regions that are no further than three lines apart (column "Fuzzy"); columns "Snippet" and "Hash" refer to the snippet-based matching and hash-based matching algorithms, respectively.

As one might expect, the overwhelming majority of matchings are exact: usually, a single commit only touches a small number of files, so almost all violations remain unaffected. Most of the remaining matchings are found by the fuzzy matching algorithm, which applies if there were changes within the same file that do not affect the violation itself, but only shift it by a few lines. Snippet-based matching only applies in a few cases, while the contribution of the hash-based algorithm varies considerably between projects: it contributes very little on Hadoop, but is more important than the fuzzy matching algorithm on Gaia.[7]

This suggests that exact matching may, in practice, be enough for many applications. For our use case of attributing violations to developers, however, this is not so: on MongoDB, for instance, hash-based matching contributes 21239 matchings. This number is vanishingly small when compared to the total number of matched violations, but it is ten times the total number of fixed violations identified across all attributed snapshots. Without hash-based matching, each missing matchings would give rise to one new violation and one fixed violation, dramatically influencing the overall result.

[7]Recall that the different algorithms are applied in stages, where more sophisticated algorithms are only run for those violations that could not be matched using the simpler algorithms.

As for **EC2**, it can ultimately only be answered by domain experts manually examining a large number of violation matchings and deciding whether they are reasonable or not. In lieu of a large-scale experiment (the outcome of which would, in the light of [21], be of doubtful significance) we manually inspected 100 randomly selected hash-based matchings from each of our four subject programs. Most of these 400 matchings corresponded to file renames or moves that were confirmed either by the snapshot's commit message or by version control metadata. The remainder were due to code being copied between files, or moved within a file with minor changes being performed at the same time, thus preventing the violations from being matched up by the snippet-based algorithm. None of the matchings were obviously wrong.

We have not yet performed a comprehensive performance evaluation of our violation matching approach. However, we observed during our experiments that the location-based and snippet-based algorithms each take about three to four milliseconds to compare two files. The hash-based algorithm is global and hence more expensive, taking around four seconds to compute matchings for a pair of snapshots.

C. Evaluating fingerprinting

The other main focus of our evaluation is to examine how meaningful our violation fingerprints are. Recall that fingerprints are computed from a set of attributed snapshots. If fingerprints for the same developer vary wildly across different sets of snapshots, we would have to conclude that they are not well-defined. Conversely, if different developers are assigned very similar fingerprints, this would mean that fingerprints are not characteristic. Finally, even if violation fingerprints are characteristic, we have to show that they do not simply reflect more basic characteristics of a developer such as average churn per commit, or average number of violations per commit.

We distill these considerations into three evaluation criteria:

EC3 Are fingerprints stable across different snapshot sets?

EC4 Is there a measurable difference between the fingerprints computed for different developers?

EC5 Are violation fingerprints independent of churn and total number of new and fixed violations?

To answer these questions, we designed an experiment in which we take two sets A and B of snapshots and compute for every developer d a violation fingerprint $f(d, A)$ based on the snapshots in set A (the training set), and a violation fingerprint $f(d, B)$ based on the snapshots in set B (the test set). Now we compare $f(d, A)$ against the fingerprints $f(d', B)$ of *all* developers (including d) as computed from set B and rank them by their Euclidean distance $\|f(d, A) - f(d', B)\|$ from the fingerprint of d.[8]

If fingerprints are highly dependent on the set of snapshots used to compute them (**EC3**), we would expect the outcome of this ranking to be mostly random. Similarly, if all developers tend to have similar fingerprints (**EC4**), a random outcome would be expected.

To address **EC5**, we perform our experiments with two kinds of fingerprints: violation density fingerprints and violation vector fingerprints scaled by total violations. The former are two-element vectors containing the total number of new violations and the total number of fixed violations, scaled by the number of lines changed. The latter are vectors with one element per violation type, scaled by the total number of fixed/new violations as described in Section V. If developers, on average, introduce and fix the same number of violations per line of code they touch, the ranking using violation density fingerprints should be random, since these fingerprints are scaled by churn. Similarly, if developers introduce and fix different kinds of violations with the same frequency, the violation vector fingerprints would produce random rankings, since they are scaled by the total number of violations.

Care has to be taken in selecting the snapshot sets A and B and in choosing which developers to compute fingerprints for. We exclude commits with less than 10 lines of churn (since they most likely have too few fixed or new violations to be interesting), and commits with more than 10000 lines of churn (since they are not likely to actually be the work of a single developer). We also do not include commits by developers who have contributed fewer than 50 commits or less than 5000 lines of churn, since their contributions are likely too small to derive a significant fingerprint from.

Overall, our experiment comprises the following steps:

1) Randomly partition the set of all considered snapshots into two halves A and B.
2) Compute fingerprints for all developers on A and B, and compute ranks as explained above.
3) For every kind of fingerprint, count how often a developer's fingerprint was ranked as being closest to themselves, second-closest to themselves, and so on.

To enhance statistical significance, we perform these steps 100 times on every test subject and aggregate the counts. The results of these experiments are shown in Figure 5, Figure 6, Figure 7, and Figure 8. Every figure contains two histograms, showing the results for the violation density fingerprint on the left, and for the violation vector fingerprint on the right. The individual bars show how often (over the 100 runs of the experiment) a developer was ranked as being closest to themselves, second-closest to themselves, and so on.[9]

We note that none of the experiments yield a random distribution. Instead, both kinds of fingerprints consistently rank developers as similar to themselves and dissimilar to others, thus suggesting a positive answer to **EC3** and **EC4**.

Since this is, in particular, true for the violation density fingerprints on all four subject programs, we conclude that the ratio of introduced and fixed violations per lines of changed code is not the same for all developers. The results are even more striking for the violation vector fingerprints: our experiments give a strong indication that each developer has

[8]Recall that fingerprints are pairs of vectors; to determine their distance, we simply concatenate the two constituent vectors into one larger vector, and then compute their Euclidean distance.

[9]Note that for readability the two graphs are not on the same scale.

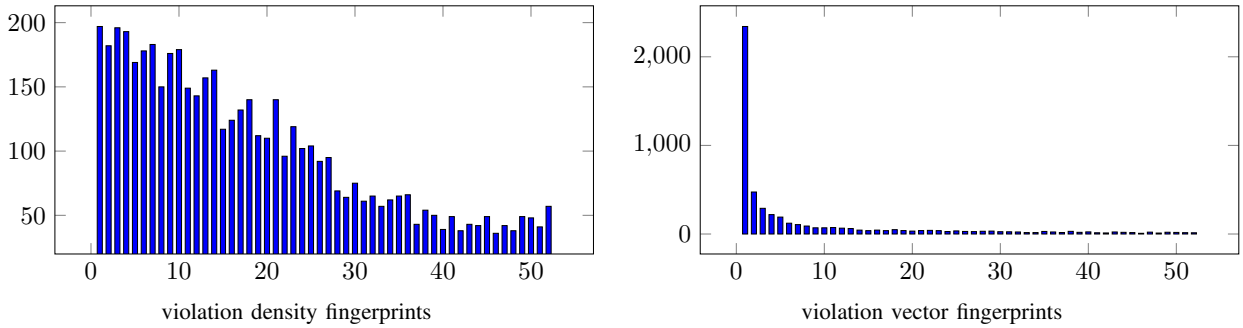

Fig. 5. Ranking results for Hadoop Common

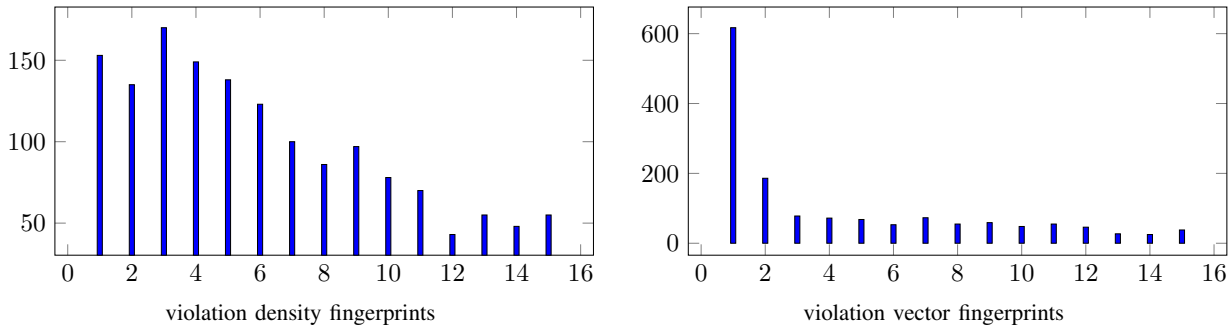

Fig. 6. Ranking results for MongoDB

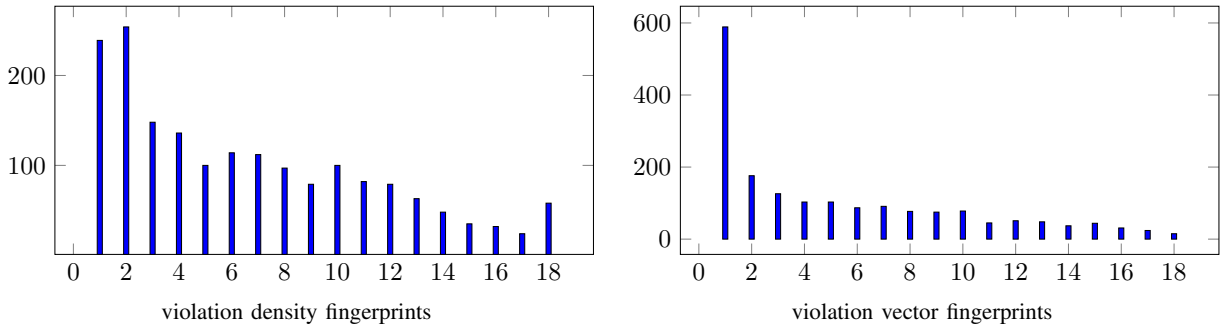

Fig. 7. Ranking results for Spark

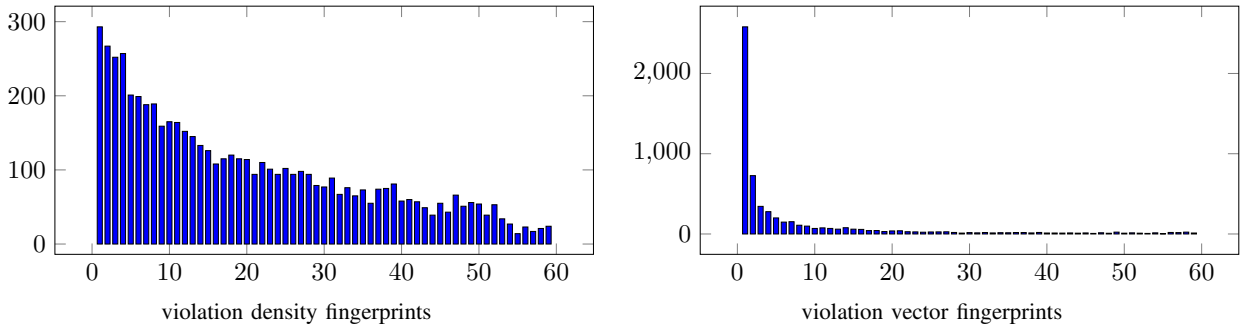

Fig. 8. Ranking results for Gaia

a very characteristic signature in terms of the violations they introduce and fix as a proportion of their overall number of violations. **EC5** can hence also be answered in the affirmative.

These results hold for all our subject programs. The comparatively weak results for MongoDB and Spark are most likely due to the relatively sparse data: across the snapshots we analysed, there were only 15 developers on MongoDB and 18 on Spark who accumulated enough churn and commits to be considered in our experiments (compared to 52 on Hadoop and 59 on Gaia), which makes the results less stable.

D. Threats to validity

Our static analysis examines source code as it is seen by the compiler during a build, including generated code. Attributing violations in generated code code is, however, not straightforward, so we manually excluded it from consideration on a best-effort basis. Given the size and complexity of our subject programs, we may have missed some files, which could affect our results. Another difficulty are bulk imports of third-party code. Our experiments account for this by excluding revisions with large amounts of churn, but this is only a heuristic.

Furthermore, we ran our static analysis with its standard rule set. Using different rules might conceivably lead to a different outcome, but preliminary experiments with subsets of the standard rule set yielded similar conclusions.

In computing fingerprints, we only consider developers who have contributed at least 50 commits and 5000 lines of churn. We have not yet experimented with varying these thresholds.

We used violation density fingerprints and violation vector fingerprints in our experiments. Many other ways of computing fingerprinting could be devised, and some of them may well be even more characteristic of individual developers than the ones we used. For example, a syntactic fingerprint based on preferred indentation style would probably be highly characteristic. However, such shallow fingerprints do not yield much insight into the contributions a developer makes to the software quality of a code base, which is our main interest.

Finally, we note that care has to be taken in generalising our results to other projects. We deliberately chose diverse projects utilising different programming languages. Although they are open source, all four projects have core developers from large software companies. Therefore, we expect our results to generalise to both commercial and open source projects.

VII. Related Work

Spacco et al. [33] discuss two location-based violation matching techniques used in FindBugs [28] and Fortify [7]. To allow for code movement, they relax their location matching in various ways, for example by matching violations at different locations within the same method, and by only considering the name (but not the full path) of the enclosing file. As in our approach, they prioritise stricter matching criteria over more relaxed ones. However, their approach cannot match violations that have moved between methods or even classes.

Diff-based matching of code snippets has been used to track violations forward [3] and fixed bugs backward [20] through revision history, as well as for tracking code clones [19]. Śliwerski et al. [32] directly use revision control metadata to pair up fix inducing commits with later bug fixes. Kim et al. [22] improve upon this by adding a form of diff-based matching similar to our approach.

Hash-based matching does not seem to have been used for tracking static analysis violations before, but the technique itself is well established in multi-version analysis [18]. Clone detectors, in particular, often use hashes to identify potential code clones within a given code base. Our hashes are token-based, similar to the one used by PMD's code clone detector [30]. Other systems use more advanced similarity metrics based on the AST [2] or the PDG [23], which are less scalable.

In origin analysis [12], [13], whole functions are hashed based on attributes such as cyclomatic complexity as well as call graph information. Our methods for violation matching strive to be language and analysis independent, and hence cannot directly employ such advanced hash functions.

Violation fingerprints as defined in this paper appear to be novel. Previous work has considered other developer characteristics such as code ownership, that is, how much experience a developer has with a piece of code [11], [14]. Typically, these characteristics are computed entirely from source control information without any static analysis.

Developer fingerprints based on layout information, lexical characteristics and software metrics have been employed in software forensics to identify authors of un-attributed code [5], [24], [34]. Spafford et al. [34] suggest considering typical bugs as well. Judging from our dataset, this seems difficult, since most revisions introduce or fix at most one or two violations, which is insufficient to derive a meaningful fingerprint.

VIII. Conclusion

We have motivated the need for enriching static analysis results with revision information to track changes in code quality over time, and attribute changes to individual developers. The main enabling technique for such an integration is violation tracking, which determines new and fixed violations in a revision relative to its parent revisions. We have discussed one approach for implementing violation tracking, and validated it on several substantial open-source projects written in different programming languages. We furthermore demonstrated that developers have characteristic fingerprints of violations they tend to introduce and fix.

It has been observed that static analysis violations are often ignored by developers [16]. Some organisations have tried to address this by imposing a commit gate that enforces adherence to coding rules, but experience with our clients has shown that this is counter-productive: sometimes business circumstances necessitate the introduction of technical debt, which manifests itself through an increased number of violations [36]. Fingerprints, on the other hand, allow each individual to keep track of their own coding habits, of where they are doing well and where they can improve, thus helping to establish an *esprit de corps* among a team of developers.

REFERENCES

[1] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5), 2008.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *ICSM*, 1998.

[3] Cathal Boogerd and Leon Moonen. Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions. In *MSR*, 2009.

[4] Coverity. Code Advisor. http://www.coverity.com, 2015.

[5] Haibiao Ding and Mansur H. Samadzadeh. Extraction of Java Program Fingerprints for Software Authorship Identification. *Journal of Systems and Software*, 72(1), 2004.

[6] Alex Eagle and Eddie Aftandilian. error-prone. https://code.google.com/p/error-prone, 2015.

[7] HP Fortify. Static Code Analyzer. http://fortify.com, 2015.

[8] Apache Foundation. Hadoop. http://hadoop.apache.org, 2015.

[9] Apache Foundation. Spark. http://spark.incubator.apache.org, 2015.

[10] Mozilla Foundation. Gaia. https://github.com/mozilla-b2g/gaia, 2015.

[11] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How Developers Drive Software Evolution. In *IWPSE*, 2005.

[12] Michael W. Godfrey and Qiang Tu. Tracking Structural Evolution Using Origin Analysis. In *IWPSE*, 2002.

[13] Michael W. Godfrey and Lijie Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE TSE*, 31(2), 2005.

[14] Lile Hattori, Michele Lanza, and Romain Robbes. Refining Code Ownership with Synchronous Changes. *ESE*, 17(4-5), 2012.

[15] James W. Hunt and Thomas G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *CACM*, 20(5), 1977.

[16] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *ICSE*, 2013.

[17] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE TSE*, 28(7), 2002.

[18] Miryung Kim and David Notkin. Program Element Matching for Multi-Version Program Analyses. In *MSR*, 2006.

[19] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. An Empirical Study of Code Clone Genealogies. In *FSE*, 2005.

[20] Sunghun Kim and Michael D. Ernst. Which Warnings Should I Fix First? In *FSE*, 2007.

[21] Sunghun Kim, Kai Pan, and E. James Whitehead, Jr. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *WCRE*, 2005.

[22] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead Jr. Automatic Identification of Bug-Introducing Changes. In *ASE*, 2006.

[23] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In *SAS*, 2001.

[24] Ivan Krsul and Eugene H. Spafford. Authorship Analysis: Identifying The Author of a Program. *Computers & Security*, 16(3), 1997.

[25] MongoDB, Inc. MongoDB. http://www.mongodb.org, 2015.

[26] Eugene W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1, 1986.

[27] Nachiappan Nagappan and Thomas Ball. Static Analysis Tools as Early Indicators of Pre-Release Defect Density. In *ICSE*, 2005.

[28] University of Maryland. FindBugs. http://findbugs.sourceforge.net, 2015.

[29] H.M. Olague, L.H. Etzkorn, S. Gholston, and S. Quattlebaum. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE TSE*, 33(6), 2007.

[30] PMD Source Code Analyzer. http://pmd.sf.net, 2015.

[31] Semmle. Project Insight. http://semmle.com, 2015.

[32] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? In *MSR*, 2005.

[33] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking Defect Warnings Across Versions. In *MSR*, 2006.

[34] Eugene H. Spafford and Stephen A. Weeber. Software Forensics: Can We Track Code to its Authors? *Computers & Security*, 12(6), 1993.

[35] Ramanath Subramanyam and M. S. Krishnan. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE TSE*, 29(4), 2003.

[36] Antonio Vetro'. Using Automatic Static Analysis to Identify Technical Debt. In *ICSE*, 2012.

[37] Antonio Vetro', Maurizio Morisio, and Marco Tochiano. An Empirical Validation of FindBugs Issues Related to Defects. In *EASE*, 2011.

[38] Jiang Zheng, Laurie A. Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. On the Value of Static Analysis for Fault Detection in Software. *IEEE TSE*, 32(4), 2006.