Fault-aware Fingerprinting: Towards Mutualism between Failure Investigation and Statistical Debugging

Chao Liu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
chaoliu@cs.uiuc.edu

ABSTRACT

Failure investigation and statistical debugging are two critical components in automated failure analysis. Although related, the two components have always been studied separately in previous work. In this paper, we propose a fault-aware fingerprinting technique that relates the two components and puts them into mutualism, *i.e.*, each component benefits and benefits from the other. Technically, we use statistical debugging algorithms to find the fault location *each* failing trace suggests, and take the suggested fault location as the fingerprint of the failing trace. The fingerprint is *fault-aware* in that it embodies the fault location each failing trace suggests. With fingerprinting, we can cluster together failing traces that suggest similar fault locations, rather than those that are literally similar to each other as done by previous work. We observe from experiments that the fingerprint-based clustering renders more meaningful results than the literal trace similarity-based approach. On the other hand, we also observe that the investigation of failing traces in the fingerprint space can help developers interpret statistical debugging results. In this paper, we report on the achieved results that exemplify the promises of the fault-aware fingerprinting technique, and propose future work for discussion and comments.

Advisor: Dr. Jiawei Han. hanj@cs.uiuc.edu.

Keywords

Fault-aware fingerprinting, Failure investigation, Statistical debugging, Failure proximity, Debugging aids.

1. MOTIVATION

Recent complex software systems, like Netscape/Mozilla, Microsoft Windows and GNOME, usually feature a failure reporting component, which, with the user's permission, automatically collects and sends back failing traces for future analysis. In principle, these collected failing traces could be used for fault diagnosis. But as collected failing traces are usually due to multiple faults, developers need to first cluster failing traces likely due to the same fault together, and then assign each cluster of failing traces to responsible developers accordingly. As failing traces can be automatically collected in large volumes, automated algorithms are needed.

Podgurski et al. propose to cluster failing traces based on the *literal* trace similarity [16]. For example, traces with similar branch and statement coverage are grouped together. Their hypothesis is that failing traces due to the same fault likely exhibit similar program behaviors. But as will be seen in Section 4, because a fault can be triggered by different inputs in different ways, the resultant program behavior can be quite divergent. Therefore, new algorithms are needed to cluster traces due to the same fault together, rather than those with similar behaviors. We denote this the *clustering* problem, a central problem in failure investigation.

Ultimately, the collected failing traces are used for fault diagnosis. Recent years have seen a number of fault localization algorithms developed [2, 6, 9, 10, 13, 15, 17–19], which can automatically identify likely fault locations. According to a recent study [11], statistical debugging algorithms, represented by Tarantula [9], Liblit05 [10] and Sober [13], are generally more accurate than the others. However, despite the accuracy, people usually complain about the *interpretability problem* of statistical debugging. Specifically, statistical debugging generates a ranking of all instrumented predicates as the debugging result. However, *without a concrete failing case*, a developer may find it hard to understand why certain predicates are ranked high. Therefore, in order to take advantage of the high accuracy of statistical debugging, a concrete failing trace that can illustrate the debugging result needs to be found.

Although both failure investigation and statistical debugging are related to fault analysis, they two have always been studied separately in previous work. In this paper, we propose a fault-aware fingerprinting technique that relates them, and furthermore puts them into mutualism. The motivation is that *now that we want to cluster failing traces due to the same fault together, why do not we automatically find the fault location each failing trace suggests through statistical debugging algorithms, and then cluster failing traces based on the suggested locations?* We take the fault location each failing trace suggests as the fingerprint of the failing trace, and investigate failing traces in the fingerprint space, rather than in the original trace space. Because

the fingerprint embodies the suggested fault location (fault-awareness), clustering in the fingerprint space will likely render more reasonable results. Moreover, because the debugging result from statistical debugging is in the same form as fingerprints, a proper failing trace that can illustrate the debugging result can be found in the fingerprint space. Therefore, by virtue of the fingerprinting technique, the clustering problem in failure investigation and the interpretability problem in statistical debugging can be tackled simultaneously.

The rest of the paper is organized as follows. We first discuss related work in Section 2, and then present the fault-aware fingerprinting technique in Section 3. We report on achieved results and propose future work in Sections 4 and 5, respectively. Finally, Section 6 concludes this proposal.

2. RELATED WORK

First, statistical debugging is related to fault localization in general. A great number of localization algorithms have been developed recently [2, 6, 9, 10, 13, 15, 17–19]. Despite their different appearances, these algorithms share a common debugging principle, that is, to contrast failure traces against passing ones. Depending on whether one or multiple failing traces are used in contrast, existing algorithms can be grouped into the following two categories.

The first category includes algorithms that contrast a given failing trace to a passing one, as represented by Delta Debugging [18], its derivatives CT [2] and HDD [15], NN/PERM [17], SLICECHOP [6] and PREDSWITCH [19]. The passing trace can be either given, like in Delta Debugging, CT and HDD, or automatically derived, like in NN/PERM, SLICECHOP and PREDSWITCH. For example, the passing trace used by NN/PERM is derived by searching a set of available passing traces, and that used by SLICECHOP and PREDSWITCH is simply the failing trace when the failure is avoided under certain manipulations. On the other hand, the second category mainly consists of statistical debugging algorithms, which contrast multiple failing traces against a set of passing ones. Representative algorithms are TARANTULA [9], LIBLIT05 [10] and SOBER [13].

Because statistical debugging bases its fault analysis on multiple failing traces, the interpretability problem naturally arises: It is unclear which failing trace can illustrate the debugging result. In comparison, algorithms in the first category do not suffer from the interpretability problem because whatever debugging result is derived from the given failing trace. Although short on interpretability, statistical debugging is shown to be more accurate than those in the first category [11]. Therefore, if the interpretability problem can be alleviated or solved, statistical debugging will be more appealing.

The second line of related work is around automated analysis of failure behaviors, which includes the following two major problems: (1) how to distinguish failing traces from passing ones, and (2) how to identify failing traces due to the same fault from a set of failing traces. For a higher level of automation, no specification is usually assumed. Previously, researchers applied classification techniques to the first problem, and encouraging results have been observed [1, 7]. Besides classification-based techniques, Dickinson et al. propose a clustering-based approach, called *cluster filtering*, which relates clustering outliers to failing traces in a set of mostly passing executions [3, 4]. As to the second problem, the most remarkable approach is proposed by Podgurski et al. [16], which clusters failing traces based on the literal trace similarity. So far, we have not seen any work that utilizes existing fault localization algorithms to better understand and analyze failure behaviors.

3. FAULT-AWARE FINGERPRINTING

Suppose a program $\mathcal{P}$ is instrumented with L predicates, and a set of m failing traces $F = \{f_1, f_2, \cdots, f_m\}$ and a set of n passing traces $P = \{p_1, p_2, \cdots, p_n\}$ are collected. Conventionally, a statistical debugging algorithm, denoted as SDA, takes P and F as inputs, and produces a predicate ranking τ as the fault debugging result, *i.e.*, $\tau = \text{SDA}(F, P)$. We call τ the *global ranking* because it is derived from all the available failing traces. In general, higher ranked predicates are more likely to be *fault-relevant*, *i.e.*, pointing to the fault location or its vicinity. Let $\tau(i)$ denote the *rank* of the predicate P_i in τ. We say predicate P_i ranks *higher* or *before* predicate P_j in τ if and only if $\tau(i) < \tau(j)$. For example, $\tau(3) = 1$ means that the predicate P_3 is the highest and its rank is one.

As one may have noticed, SDA does *not* require F and P in their entirety. Instead, any subsets of F and P can be contrasted through SDA. As an extreme case, we can contrast each failing trace $f_i \in F$ against P, and obtain the debugging result τ_i, *i.e.*,

$$\tau_i = \text{SDA}(\{f_i\}, P) \quad (i = 1, 2, \cdots, m). \tag{1}$$

τ_i is called an *individual ranking*, and is taken as the fingerprint of f_i. Moreover, because τ_i shows the fault location f_i suggests, it is *fault-aware*.

Now that failing traces are fingerprinted into predicate rankings, we can assess the proximity between failing traces (*i.e.*, failure proximity) in the fingerprint space. A distance function Dist is needed to define on pairs of predicate rankings, such that $\mathsf{Dist}(\tau_i, \tau_j)$ is small if and only if f_i and f_j roughly suggest the same fault location. Once such a Dist function is decided, the pair-wise distance between failing traces can be calculated, based on which failing traces can be clustered accordingly. Because failing traces suggesting roughly the same fault location are clustered together, we hope this fingerprint-based approach could render better clustering results. Finally, we note that because τ is also a predicate ranking, its distance from individual rankings can also be calculated with the same Dist function. This makes it possible to find a proper failing trace to illustrate the debugging result τ.

In comparison with previous methods that group failures based on the *literal* trace similarity, this fingerprint-based approach features following advantages.

- Clustering in the fingerprint space are more likely to group failing traces due to the same fault together by virtue of the fault awareness of fingerprinting.
- Since only failing traces suggesting similar fault locations are clustered together, thus formed clustering automatically provides a guess about the fault location the traces in each cluster are due to. Specifically, the predicates ranked high in most individual rankings of each cluster is the guessed fault location. This could guide the assignment of failing traces to the responsible developers, a task that might otherwise requires non-trivial manual work.

- Because the distance between τ and τ_i's can be calculated, the failing trace whose fingerprint is the closest to τ is a proper failing trace to illustrate the debugging result.

As one can see, the last two advantages cannot be accomplished through trace similarity-based clustering. In the next section, we report on our achieved results, which support the above three advantages.

4. ACHIEVED RESULT

In 2005, we developed a statistical debugging algorithm, called SOBER, which automatically sifts fault-relevant predicates based on a similar reasoning as hypothesis testing. Specifically, every instrumented predicate is pre-assumed innocent (*i.e.*, irrelevant to fault) until a great divergence has been observed between its behavior in failing and passing traces. Readers interested in the technical details of SOBER are referred to [13] and [11].

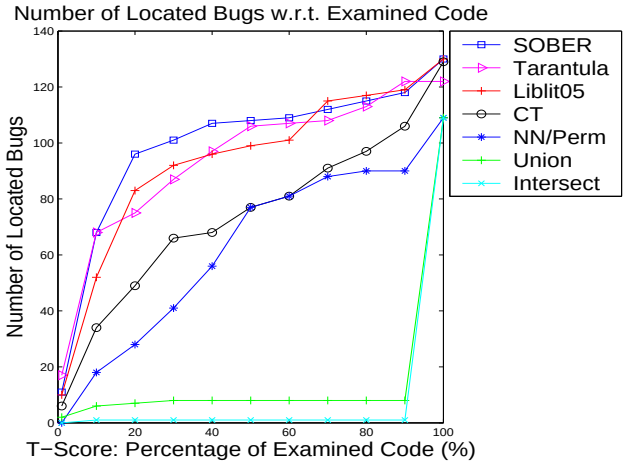

Figure 1: Localization Quality Comparison

The SOBER algorithm proves effective in fault localization. Figure 1 shows the quality comparison between SOBER and another six existing localization algorithms, as evaluated on the Siemens suite [8]. The Siemens suite contains 130 faulty versions of seven subject programs, where each faulty version contains one and only one manually injected fault. The x-axis is for the T-score, which estimates what percentage of code needs to be examined (before locating the fault) given a debugging result. The y-axis indicates how many faults can be located by each algorithm under a specific T-score. The algorithms PREDSWITCH [19], HDD [15] and SLICECHOP [6] are not depicted in Figure 1 because no result on the Siemens suite is available for PREDSWITCH and HDD, and only 38 faulty versions were used in the evaluation of SLICECHOP [6]. As indicated by Figure 1, SOBER is clearly among the most effective fault localization algorithms. We also compared SOBER with SLICECHOP on their used 38 versions, and SOBER was shown better [11].

Besides SOBER, we recently implemented the fault-aware fingerprinting technique, and had initially verified its usage [12]. Specifically, we instantiated SDA with the SOBER algorithm, and Dist with a weighted Kendall's tau distance. For clarity in what follows, we define the failure proximity

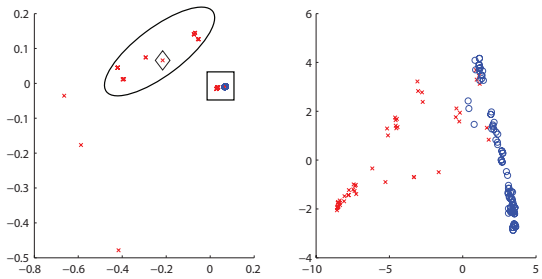

Figure 2: Proximity Graphs with R-Proximity (left) and T-Proximity (right)

calculated in the fingerprint space as R-PROXIMITY, and refer to the proximity based on literal trace similarity [16] as T-PROXIMITY.

We evaluated the advantages of R-PROXIMITY over T-PROXIMITY through a case study on `grep`-2.2, where two logic faults were manually injected. The first fault is an "off-by-one" error injected at Line 533 in `grep.c`, and the other one is a "subclause-missing" error at Line 2270 in `dfa.c`. The two faults together failed 136 of the entire 480 test cases (48 due to the first fault and the other 88 due to the second). Figure 2 visualizes the proximity between the 136 failing cases under R-PROXIMITY (left) and T-PROXIMITY (right). The red crosses and blue circles represent the failing cases due to the first and the second fault, respectively, and the red cross bounded by a diamond symbol denotes the global ranking.

Supportive results for the claimed three advantages are observed from this case study. First, failing traces tend to form denser clusters under R-PROXIMITY, as indicated by the rectangular region in the left figure where failing traces due to the second fault form a dense cluster. In contrast, these traces stretch in a line under T-PROXIMITY. This observation indicates that failing traces due to the same fault can actually exhibit quite divergent behaviors, which explicitly undermines the hypothesis held by T-PROXIMITY. Secondly, by examining the top predicates of the constitute rankings, failing traces in each cluster can be assigned to responsible developers in a guided way. For example, as most rankings in the rectangular region rank a predicate pointing to Line 2270 of the `dfa.c` file at the top, the corresponding failing traces should be assigned to the developers in charge of the `dfa.c` file. Finally, a failing trace whose fingerprint is the closest to the global ranking is easily found under R-PROXIMITY. And this trace proves helpful to help developer interpret the debugging result. Readers interested in more experimental details are referred to [12].

5. PROPOSAL OF FUTURE WORK

Because the fault-aware fingerprinting technique provides a brand-new presentation of failing traces, we believe its usage has not been fully exploited. In this section, we put forward three pieces of future work for discussion.

5.1 Fault Localization as Election

Because fingerprint rankings embody the fault each failure trace suggests, we now consider an *election*-like approach to fault localization. Specifically, the instrumented predicates are the candidates, and each failing trace is like a voter. In consequence, the fingerprint rankings are the votes for fault

locations: higher ranked predicates (*i.e.*, locations) receive a larger weight of favor. If we can consolidate the m votes $\tau_1, \tau_2, \cdots, \tau_m$ properly, better fault debugging results can be expected by the virtue of election.

The problem of consolidating multiple rankings is called *rank aggregation*, and has been well-studied in both statistics and computer science [5,14]. But no one has explored its usage in fault localization. Moreover, based on our understanding, some widely-adopted rank aggregation algorithms, like MedRank [5], may not immediately apply to fault localization because some factors specific to fault localization need to be considered. For example, the proximity between predicates should be taken into account, and some superfluous predicates may need to be first excluded before aggregation.

5.2 PDG-based Proximity Measurement

Although encouraging results have been obtained with R-Proximity, where the Dist function is instantiated with a weighted Kendall's tau distance, there could be other distances to better quantify the extent to which two fingerprint rankings suggest the same fault. In R-Proximity, the proximity between predicates is not considered, which could be a potential drawback. For example, suppose P_1 and P_2 are two predicates pointing to two different but close locations. Then under R-Proximity, two rankings with P_1 and P_2 as the top predicate respectively are regarded dissimilar although they do suggest the same location. Therefore, a distance function that takes predicate proximity into account will likely perform better.

Inspired by how T-score is defined to measure the proximity between debugging results and the real fault location [13], we propose to define Dist based on Program Dependency Graphs (PDGs). First, the top-k predicates of each ranking are treated as the *set* of fault locations suggested by the corresponding failing trace. Then the distance between two failing traces is defined by the closeness between the two sets of locations on PDGs. For example, we can expand the two sets of locations simultaneously through breadth-first search, and check what percentage of the entire PDG is covered before the expanded sets intersect. In comparison with R-Proximity, this PDG-based approach incorporates PDGs as a knowledge base for distance calculation, and in consequence, is expected to characterize the failure proximity more truthfully.

5.3 Beyond Statistical Debugging

Although our discussion has been around statistical debugging so far, the fingerprinting idea is actually compatible with other debugging algorithms. The only requirement for a given debugging algorithm is a distance definition that properly quantifies the closeness between two debugging results. Now that a great number of fault localization algorithms have been developed, it could be interesting to investigate what distance is needed by each algorithm, and which algorithm best fingerprints failing traces for clustering.

6. CONCLUSION

In this paper, we proposed a fault-aware fingerprinting technique that relates statistical debugging and failure investigation, and furthermore puts them into mutualism. The statistical debugging-based fingerprinting technique renders more meaningful clusterings of failing traces, and failure investigation in the fingerprint space helps alleviate the interpretability problem of statistical debugging. We reported on the achieved results, and proposed three pieces of future work for discussion.

Because our background is mainly on statistics and data mining, we are eagerly seeking comments and guidance from experts in software engineering community. For example, we are unclear what future work is really worthwhile, and would like to know how industry processes the huge volume of collected failing traces .

7. REFERENCES

[1] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA'04*.

[2] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE'05*.

[3] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE'01*.

[4] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *ESEC/FSE'01*.

[5] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW'01*.

[6] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE'05*.

[7] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely collected program execution data. In *ESEC/FSE'05*.

[8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE'94*.

[9] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE'05*.

[10] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI'05*.

[11] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transaction on Software Engineering*, 32(10), 2006.

[12] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *FSE'06*.

[13] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *ESEC/FSE'05*.

[14] J. I. Marden. *Analyzing and Modeling Rank Data*. Chapman & Hall/CRC, first edition, 1996.

[15] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *ICSE'06*.

[16] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE'03*.

[17] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE'03*.

[18] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE'02*.

[19] X. Zhang, R. Gupta, and N. Gupta. Locating faults through automated predicate switching. In *ICSE'06*.