# RAPID: Identifying Bug Signatures to Support Debugging Activities

Hwa-You Hsu, James A. Jones, and Alessandro Orso
College of Computing
Georgia Institute of Technology
{hsu|jjones|orso}@cc.gatech.edu

## Abstract

*Most existing fault-localization techniques focus on identifying and reporting single statements that may contain a fault. Even in cases where a fault involves a single statement, it is generally hard to understand the fault by looking at that statement in isolation. Faults typically manifest themselves in a specific context, and knowing that context is necessary to diagnose and correct the fault. In this paper, we present a novel fault-localization technique that identifies sequences of statements that lead to a failure. The technique works by analyzing partial execution traces corresponding to failing executions and identifying common segments in these traces, incrementally. Our approach provides developers a context that is likely to result in a more directed approach to fault understanding and a lower overall cost for debugging.*

## 1 Introduction

Debugging is an expensive and time-consuming activity that can be responsible for a significant part of the cost of software maintenance. The use of automated or semi-automated techniques for supporting debugging has therefore the potential to be extremely beneficial in reducing the overall cost of software development and, at the same time, improving software quality.

There has been a great deal of research in the area of automated debugging in recent years (*e.g.,* [2,4,6,13,14]). These novel techniques have been shown to be effective and have greatly contributed to advancing the state of the art. However, most of these debugging approaches suffer from an important limitation: they typically assume *perfect bug understanding*—simply examining a faulty statement in isolation is always enough for a developer to detect, understand, and correct the corresponding fault.[1]

In some cases, however, single lines of code do not provide enough information. Obviously, this is the case for faults that involve multiple, possibly far apart, statements. In our experience, even in situations where a fault involves a single statement, it is generally difficult to understand the fault by looking at that statement in isolation. Faults typically manifest themselves in a specific context, and knowing that context is necessary to diagnose and correct the fault.

Some existing approaches, such as techniques based on program slicing or on a collection of (rich) faulty execution traces (*e.g.,* [1, 3, 7, 12, 14]), provide context for the investigation of a failure. However, in most cases, these approaches provide too much information, and the relevant information gets lost in the noise. Program slices, for instance, tend to include large parts of a program, most of which are irrelevant for the investigation of a failure.

To mitigate these limitations of existing techniques, in this paper, we present a novel fault-localization technique whose goal is to find a sweet spot between providing too little and too much information to developers.

Like most existing techniques, our technique assumes the availability of a set of test cases for the program being debugged. This set includes both test cases for which the program fails and test cases for which the program behaves correctly (*i.e.,* it passes). To compute relevant information that can support debugging, our technique performs three main steps. First, it analyzes passing and failing executions to identify the likelihood of each statement to be responsible for the failure under investigation. Second, it analyzes partial failing execution traces, that is, traces that contain only statements above a given suspiciousness threshold. This analysis uses a sophisticated string matching algorithm to identify one or more common subsequences in these traces. Third, it reports the identified subsequences to the developers. If the subsequences do not contain enough information to locate and understand the fault, the technique repeats the second step using a lower suspiciousness threshold.

Compared to most existing debugging approaches, our technique has several advantages: at the same time, it (1) considers the information contained in all passing and failing executions, (2) automatically identifies relevant context for faults, when such context exists, and (3) allows for identifying such context incrementally, adding more information when the currently computed context is insufficient.

We also discuss an example taken from a preliminary experiment we performed using RAPID, a prototype tool that

[1] Some of these techniques actually consider program entities other than statements, such as branches or predicates. Because considering different entities does not change the nature of the problem, in this paper, for simplicity, we simply refer to statements as the entities of choice.

implements our technique. The example shows the effectiveness of our approach for faults that manifest due to a sequence of events.

The main contributions of this work are:

- A novel fault localization technique that identifies sequences of statements involved in a failure by analyzing partial failing execution traces and identifying common segments in these traces, incrementally.
- A tool, RAPID, that implements the different parts of our technique for C programs.
- An example of application of our approach that shows its potential usefulness.

## 2 Our Approach

As it is typically the case for fault-localization approaches, the starting point of our technique is a program to be debugged and a set of test cases for the program that includes both *passing test cases* (*i.e.,* test cases for which the program produces a correct result) and *failing test cases* (*i.e.,* test cases for which the program manifests a failure). Given this information, our technique performs three main steps: dynamic data collection, bug signature identification, and interactive localization. In the next sections, we discuss these three steps in detail and illustrate them using the intuitive view of our technique provided in Figure 1.

### 2.1 Step 1: Dynamic data collection

In this step, our technique collects the dynamic information that it uses to identify bug signatures: coverage information and execution traces. The technique instruments the program being debugged and generates an instrumented version of the program that produces, at runtime, a trace of the branches (*i.e.,* method entries and outcomes of decision statements) traversed during the execution. Collecting branch data alone is sufficient because information for all statements in the program can be easily derived from such data.

Our technique then runs the instrumented program against all passing and failing test cases and records the trace information produced by the program for each test case. It then extracts coverage information from the traces—for each test case, it checks which branches (and thus statements) were covered by that test case. Coverage information is used to compute, using a statistical ranking technique, the likelihood of each element covered to be responsible for a failure. In the rest of the paper, we call this value *suspiciousness*.

Many techniques could be used to compute suspiciousness values (*e.g.,* [6, 8, 9]). For our technique, we use a ranking technique defined by one of the authors of this paper, called Tarantula [5, 6]. Suspiciousness values for a program element $e$ (a branch, in this case) can range from 0, which indicates that $e$ is not suspicious, to 1, which indicates that $e$ is highly suspicious.

The leftmost part of Figure 1 illustrates, in a graphical fashion, the two pieces of dynamic information collected in this step. In the upper part, it shows the set of suspiciousness values attached to the branches of the program, represented as control-flow graphs. In the lower part, it shows the set of traces collected for the failing test cases, represented as a sequence of elements. (Each of these is magnified to give an example of individual suspiciousness values for branches and sequences of events for the traces.)

### 2.2 Step 2: Bug signature identification

In this step, our technique computes *bug signatures*—sequences of program elements that, when executed in order, are likely to lead to a failure. The technique first associates the previously computed suspiciousness information with the branches in the failing traces, so that each branch in a trace is labeled with its corresponding suspiciousness value. This information is then used to eliminate from the traces all elements below a given suspiciousness threshold. The resulting (possibly non-contiguous) traces are partial failing traces containing only elements that are more likely to be faulty according to the Tarantula ranking technique. To include as many elements as possible, while eliminating the ones that are fairly unlikely to be related with a failure, we choose 0.6 as a threshold. (Intuitively, elements with 0.6 suspiciousness have a 60% chance of being related to the failure.)

Our technique computes bug signatures by identifying common patterns in the filtered traces; it encodes the problem of finding such patterns in a set of traces as a longest-common-subsequence identification problem, where a *common subsequence* is a (usually non-contiguous) sequence that occurs in all traces, and a *longest common subsequence* is a common subsequence that is not contained in any other common subsequence. Among the many approaches proposed to address this problem in the data-mining community, we selected a technique called BI-Directional Extension (BIDE) [11]. To the best of our knowledge, BIDE is currently the most efficient algorithm to solve this problem.

When successful, this step produces the longest sequences of statements that (1) have a suspiciousness level greater than 0.6, and (2) occur, in the same order, in all failing executions. These sequences are used in the third step of the technique to support developers in their debugging effort.

Figure 1 illustrates Step 2 of our technique. Its upper, central part shows the set of traces annotated with suspiciousness information. (For simplicity, we represent only partial information.) The grayed-out boxes are elements with suspiciousness values below 0.6, which are filtered out, and the dashed lines between trace elements indicate the matching of these elements identified by the BIDE technique. The lower part of the figure shows the longest subsequence (*i.e.,* bug signature) identified by the technique: "1F, 2T, 3T."

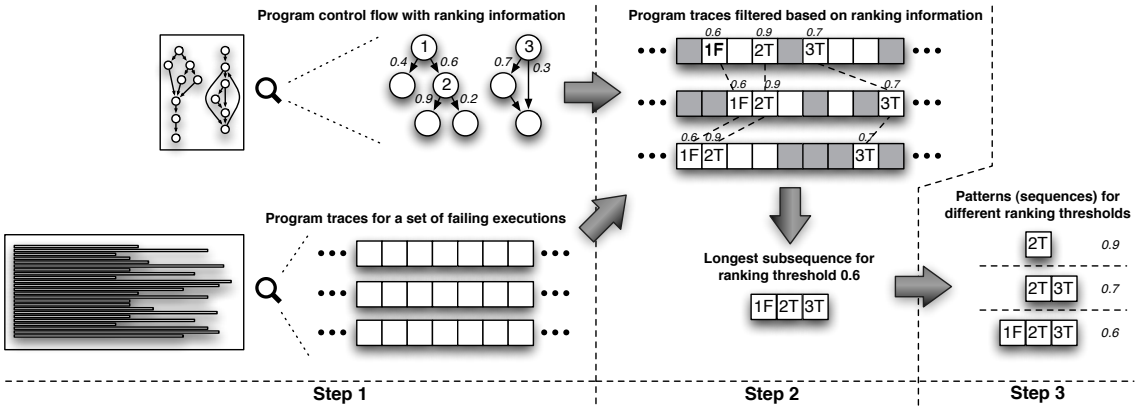


**Figure 1. High-level view of the three steps of our technique.**

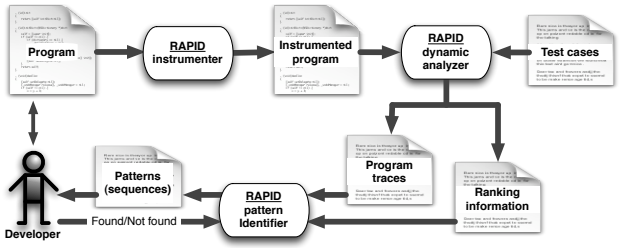


**Figure 2. Workflow in the RAPID tool.**

### 2.3 Step 3: Interactive localization

In its third step, our technique presents sequences of suspicious program elements to the developers in an iterative, incremental fashion. In the first iteration, the technique extracts from the sequences identified in the previous step the subsequences of elements with the highest suspiciousness. Such subsequences can include one or more statements, depending on the distribution of the suspiciousness values. Developers would then study the provided subsequences and, by also leveraging their knowledge of the program, try to (1) assess whether they contain the fault (or faults) causing the failures being investigated and (2) understand such fault and how to eliminate it.

If the developer is able to locate and understand the fault(s) using this first set of sequences, the technique terminates. Otherwise, it adds to the subsequences the elements with the second highest suspiciousness and presents this longer subsequences to the developer. This process continues until either the developers are successful in their investigation or the subsequences become too large to be useful.

Step 3 is illustrated in the rightmost part of Figure 1. The figure shows the three increasingly long subsequences that would be presented to developers in an incremental fashion during their investigation of the failure. As the developers require more contextual information, the threshold would be lowered to include more events in the bug signature.

## 3 Experience

To assess practicality and usefulness of our fault-localization approach, we implemented it in a prototype tool, RAPID (Ranking, Analysis, and Pattern Identification for Debugging), and used the tool on the Siemens programs and faults [10]. We show how our approach can provide better support for fault localization than other techniques using one of the bugs that we studied, for program `replace`.

Program `replace` takes three inputs—a string $s$, a pattern $p$ to be identified in $s$, and a string $r$—and replaces all occurrences of $p$ in $s$. Figure 3 shows part of function `esc`, which is in charge of recognizing and handling escape characters in $p$ and $s$. Statement 6 of `esc` contains a fault: variable `result` is assigned value ENDSTR instead of value ESCAPE. The figure also shows a fragment of function `subline`, whose purpose is to check whether there are any occurrences of pattern $p$, contained in parameter `pat`, in string $s$, contained in parameter `lin`. If an occurrence is found, `subline` replaces such occurrence with string $r$, contained in parameter `sub`.

For this example, the fault results in a failure if and only if (1) pattern $p$ contains an escape character, (2) the escape character in $p$ is followed by an `ENDSTR` character, and (3) there is at least an occurrence of `pat` in `lin`. Conditions (1) and (2) make the predicates at lines 2 and 5 in `esc` false and true, respectively, and result in the statement at line 6 being executed. Condition (3) makes the predicate at line 7 in `subline` true and results in the statement at line 8 being executed. Executing any of these two statements alone is not enough to reveal the fault. Only executions where the two statements are executed in *sequence* result in a failure.

Existing fault-localization techniques that focus on individual statements in the code are likely to be ineffective in this case. Even in the ideal case where they rank both line 6 in `esc` and line 8 in `subline` as statements with high likelihood to be related to the failure, they would have no way to correlate these statements. They would therefore present the two statements to the developer as independent entities. Moreover, there would typically be other entities with similar, or even higher, ranking. Conversely, most techniques that focus on slicing or on collecting and reporting faulty execution traces would be able to identify subsets of

```
      char esc(s, i)
       char    *s;
       int     *i;
      {
1:        char        result;
          ....
2:        if (s[*i] != ESCAPE)
3:            result = s[*i];
4:        else
5:            if (s[*i + 1] == ENDSTR)
6:                result = ENDSTR; // fault
          ...
      }

      void subline(lin, pat, sub)
       char   *lin;
       char   *pat;
       char   *sub;
      {
1:        int i, lastm, m;
2:        lastm = -1;
3:        i = 0;
4:        while ((lin[i] != ENDSTR))
5:        {
6:            m = amatch(lin, i, pat, 0);
7:            if ((m >= 0) && (lastm != m)) {
8:                putsub(lin, i, m, sub);
          ...
      }
```

**Figure 3. Excerpt of code from `replace` that shows a path-dependent fault. Although the fault is located at line 6 of function `esc`, both of the highlighted statements must be executed for the fault to be revealed.**

**Table 1. Suspiciousness data for `replace`**

| Suspiciousness | Elements |
|---|---|
| 0.99 | Branch `true`, line 5 in `esc` |
| 0.77 | Branch `true`, line 23 in `in_pat_set` |
| ··· 27 *other elements* ··· | |
| 0.63 | Branch `true`, line 7 in `subline` |
| ··· | |

the program that contain the two statements. The two statements, however, would likely be presented to the developers together with a potentially large number of additional statements, which would make it difficult to isolate the actually relevant elements.

For this version of `replace`, RAPID reports a bug signature of six elements: (1) line 2 of `esc` to be false, (2) line 5 of `esc` to be true, (3–5) three other irrelevant elements, and (6) line 7 in `subline` to be true. Compare this result with Table 1, which shows the suspiciousness values computed by Tarantula for this program and failure. Analyzing entities by decreasing suspiciousness would not result in any meaningful sequence and would require developers to examine 28 additional irrelevant entities before considering the two necessary conditions for the failure.

## 4 Conclusion

This paper presents a novel approach for helping developers find and understand faults. Our approach works by (1) collecting run-time data, (2) utilizing light-weight approaches to identify likely program regions that are responsible for causing failures, (3) identifying non-continuous sequences of events (bug signatures) that are common in all failures, and (4) providing developers with a way to explore these bug signatures.

Compared to traditional fault-localization techniques, our approach has the main advantages of automatically identifying relevant context for faults, when such context exists, and providing developers with such context incrementally, when (and if) needed. Although a more extensive evaluation of the approach is needed before drawing any conclusion, our initial results are promising. We believe that our technique can result in a more directed approach to fault understanding and decrease the overall cost of debugging.

## Acknowledgments

This work was supported in part by NSF awards CCF-0725202 and CCF-0541080 to Georgia Tech.

## References

[1] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of IEEE Software Reliability Engineering*, pages 143–151, 1995.

[2] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering*, pages 342–351, St. Louis, Missouri, May 2005.

[3] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *International symposium on Software testing and analysis*, pages 121–134, 1996.

[4] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 184–193, New York, NY, USA, 2007. ACM.

[5] J. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[6] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 467–477, Orlando, Florida, May 2002.

[7] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[8] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005.

[9] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of European Software Engineering Conference and Foundations on Software Engineering*, pages 286–295, September 2005.

[10] Laboratory for Empirically-based Software Quality Research and Development. Software-artifact Infrastructure Repository. `http://sir.unl.edu/php/index.php`, 2008.

[11] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 79, Washington, DC, USA, 2004.

[12] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[13] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[14] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 169–180, New York, NY, USA, 2006.