Identifying Bug Signatures Using Discriminative Graph Mining

Hong Cheng
Dept. of Systems Engineering and Engineering Management
Chinese University of Hong Kong
hcheng@se.cuhk.edu.hk

David Lo
School of Information Systems
Singapore Management University
davidlo@smu.edu.sg

Yang Zhou
Dept. of Systems Engineering and Engineering Management
Chinese University of Hong Kong
zhouy@se.cuhk.edu.hk

Xiaoyin Wang
Key Laboratory of High Confidence Software Technologies (Peking University)
Ministry of Education, Beijing, 100871, China
wangxy06@sei.pku.edu.cn

Xifeng Yan
Dept. of Computer Science
University of California at Santa Barbara
xyan@cs.ucsb.edu

ABSTRACT

Bug localization has attracted a lot of attention recently. Most existing methods focus on pinpointing a single statement or function call which is very likely to contain bugs. Although such methods could be very accurate, it is usually very hard for developers to understand the context of the bug, given each bug location in isolation. In this study, we propose to model software executions with graphs at two levels of granularity: methods and basic blocks. An individual node represents a method or basic block and an edge represents a method call, method return or transition (at the method or basic block granularity). Given a set of graphs of correct and faulty executions, we propose to extract the most discriminative subgraphs which contrast the program flow of correct and faulty executions. The extracted subgraphs not only pinpoint the bug, but also provide an informative context for understanding and fixing the bug. Different from traditional graph mining which mines a very large set of frequent subgraphs, we formulate subgraph mining as an optimization problem and directly generate the most discriminative subgraph with a recently proposed graph mining algorithm LEAP. We further extend it to generate a ranked list of top-k discriminative subgraphs representing distinct locations which may contain bugs. Experimental results and case studies show that our proposed method is both effective and efficient to mine discriminative subgraphs for bug localization and context identification.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging; H.2.8 [Database Management]: Database Applications – Data Mining

General Terms: Algorithms, Experimentation

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

1. INTRODUCTION

Software bugs have been a part of day-to-day software development. Debugging effort has been a painstaking and expensive task to software developers. It is particularly hard to identify, locate and fix a bug given known failures in a system, as the bug can appear far from the point where a failure becomes observable (e.g., when a computation result is outputted, when the system crashes, etc.). Certainly, automating part of this debugging process can substantially help in both easing the programmers' effort in locating bugs and reducing the overall cost of software development. A past study has estimated that the cost of debugging can go up to billions of dollars annually [18].

There have been many promising studies in the area of fault localization and automated debugging, e.g., [12, 17, 24, 5, 14, 15]. These studies usually take as input two sets of program traces corresponding to correct and faulty executions. A trace itself is generated by instrumenting the target program and logging *events* corresponding to method invocation, statement execution, basic block entry, etc., when the instrumented program is run. Based on these two sets of input, a set of candidate bug locations is then reported. These studies have shown that fault localization techniques are able to pinpoint the root causes of various bugs in many cases.

Most research work on fault localization assumes "perfect bug understanding" [7] – where a programmer is able to identify a bug simply by looking at a faulty line of code in isolation. Often, more than the exact location of a bug is needed, rather the "context" where the bug occurs is likely to help greatly in improving the programmers' ability in identifying, understanding and correcting bugs. For example, it might be the case that executing any one of the two statements s_1 and s_2 does not result in a fault. However, when these two statements are executed one after the other, then the fault will occur. The context of s_1 being executed before s_2 is important in aiding programmers to understand and fix the bug. Some other approaches use program slicing to provide for context [2, 6, 11, 25]. However often the slices are too large and include too much information which might potentially dilute important bug-related information among the noise.

Based on the above motivation, Hsu *et al.* [7] have developed a tool, referred to as RAPID, to identify bug signatures from correct and faulty executions of a program. A bug signature provides the context where the bug occurs. Their approach first identifies relevant suspicious statements from program executions by contrasting correct and faulty executions via Tarantula [12]. A statement is suspicious if it appears more in the faulty executions than the correct executions. Next, *based only on the faulty executions*, they compute the longest common sub-sequences that appear in *all* the faulty executions. The returned signatures are then sorted by length and presented to the user. They have shown that their approach is able to identify a bug involving the path-dependent fault which is hard to identify by past techniques that do not report contextual information.

In this paper, as a further step along this direction, we develop a discriminative graph mining algorithm for identifying bug signatures. A software execution can be "coiled" to form graphs capturing the relationship between traced events. We refer to this graph as *software behavior graph*. We consider two levels of granularity: method and basic-block. At the method level, an execution trace corresponds to method invocations; while at the basic block level, an execution trace corresponds to executions of basic blocks. The higher the level of granularity, the shorter the trace is and we are then able to analyze larger systems. The trade-off is that at a higher level of granularity we have a less rich input which potentially reduces the accuracy of a bug signature identification method. Hence, we believe that different levels of granularity have their own pros and cons. It would be interesting to analyze the effect of different levels of granularity on bug signature identification results.

A behavior graph is composed of a set of nodes each corresponding to a method or basic block, and a set of edges each corresponding to a relationship (call, return or transition) between the respective pair of nodes. Our proposed graph mining approach will extract the most discriminative subgraphs which are highly indicative of bugs and their contexts from behavior graphs of correct and faulty runs. The advantages of our graph mining approach over a sequence based approach include: (1) a graph-based representation compactly summarizes a long execution trace while a sequence representation can grow very long due to loops; (2) a discriminative graph pattern increases the expressive power of the longest common sub-sequence used by RAPID, as we can now express both partial and total order in one representation, while a sub-sequence on the other hand can only express total order; (3) our method relaxes the requirement that a bug signature must appear in *all* faulty executions. This is beneficial since a bug can have a number of representations and be observed with different signatures in the faulty executions; and (4) while RAPID only finds discriminative single events via Tarantula, we look for discriminative features involving multiple events.

Intuitively, a bug will cause structural differences between the software behavior graphs of faulty and correct runs. As a pre-processing step, we filter off non-discriminative edges from the graph. These edges correspond to relationships between methods (or basic blocks). Different from RAPID, we filter off non-suspicious relationships between nodes rather than the nodes themselves. In our experiments, we find that this is more effective in producing better quality bug signatures as we can retain suspicious relationships between two non-suspicious nodes. Next, we perform graph mining on the graph dataset to extract discriminative subgraphs which are very likely to be bug relevant.

Our discriminative graph mining approach is different from traditional frequent graph mining, e.g., [23, 16], in the following aspects: (1) traditional graph mining methods take as input a single set of graphs without labels. The whole set of frequent subgraphs are then generated wrt. a user-specified minimum frequency threshold, but many of them may not be related to bugs. In contrast, we take two sets of graphs as input from correct and faulty executions. We then formulate our graph mining problem as an optimization problem, i.e., to directly search for the most significant subgraph in terms of its relevance to bugs. As a result, our method is much more effective and efficient in localizing bugs; and (2) we further extend our method to generate a ranked list of k most suspicious subgraphs to help users localize bugs.

With the discriminative graph mining approach, we directly discover top-k discriminative subgraphs which can highlight the contrasting program flows between correct and faulty executions and preserve a partial program flow between functions/statements related to the fault. Such discriminative subgraphs serve as signatures highlighting potential bug locations and provide informative contexts where bugs occur, which in turn help to guide programmers in finding the source of bugs and correcting them.

In summary, our main contributions include:

1. We propose a bug signature mining framework with a discriminative graph mining approach. Software executions are compactly modeled as graphs. A preprocessing step is employed which filters off non-discriminative edges. We then formulate the bug localization problem as a discriminative graph mining problem. A recently proposed algorithm LEAP [22] is used to directly find the most discriminative subgraph which contrasts faulty executions from correct ones.

2. We further extend LEAP to mine top-k discriminative subgraphs, since it is more informative to generate a ranked list of multiple bug signatures than the single best one. The method is called Top-K LEAP. We find this strategy to be effective in ranking candidate bug signatures.

3. We perform preliminary experimental studies on the Siemens benchmark datasets to evaluate our method. Experimental results demonstrate that our method is effective in recovering bugs and their contexts. On average, we improve the precision and recall of RAPID by up to 18.1% and 32.6% respectively. In addition, our method is shown to be more efficient. We are able to complete subgraph mining on every dataset within 258 seconds while RAPID is not able to complete mining some datasets even after running for hours.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 defines the preliminary concepts. Section 4 introduces our mining framework and illustrates the benefit and effectiveness of our bug signature identification approach via discriminative graph mining. Section 5 formulates the bug localization problem as a discriminative subgraph mining problem, discusses the challenges in this mining task and uses a Top-K LEAP method

which mines top-k subgraphs as the solution. Experimental evaluation is presented in Section 6. Section 7 discusses some issues and future work. Section 8 concludes our study.

2. RELATED WORK

There are three closely related threads of work to our approach. These include the pioneer of the work on bug signature identification [7], many of the past studies on bug localization and past studies on graph mining. The following sub-sections describe them in more details and contrast them with our proposed bug signature identification framework via discriminative graph mining.

2.1 RAPID: Identifying Bug Signatures

RAPID is the pioneering work on bug signature identification. It mines for bug signatures using a merge between Tarantula [12] and a sequential pattern mining approach [21]. It aims to complement existing techniques pinpointing the exact single statement that may contain a bug by providing the context in which a bug occurs. This context is captured in the form of a signature that must hold across *all* faulty executions.

Similar to other approaches on bug localization, RAPID starts by accepting two sets of traces: faulty and correct. RAPID obtains traces containing statements corresponding to method entries and branch decisions. In our experimental study, we enrich the input of RAPID to include all basic block level information, including method return. RAPID then uses Tarantula [12] to measure the score of suspiciousness of statements based on the following formula:

$$suspiciousness(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}}$$

The operators $failed(s)$ and $passed(s)$ report the number of faulty program traces that include s and the number of correct program traces that include s respectively. The suspiciousness of a statement is proportional to the normalized ratio between the number of faulty traces that contain it versus the total number of correct and faulty traces that contain it.

RAPID fixes the Tarantula threshold at 0.6 (i.e., 60% likelihood of a statement being related to a failure). The remaining suspicious statements *in the faulty runs* are then collected and formatted to form a multi-set (or bag) of sequences of events (or statements). This bag of sequences are then given to the state-of-the-art sequential pattern miner BIDE [21] with a frequency threshold of 100%. With that high threshold, BIDE will return one or more longest common subsequences (LCSs) that the set of sequences have. These LCSs are then sorted according to length and presented to the user. Since the user is less likely to navigate through a large number of bug signatures, we only return top 10 results after the LCSs are sorted according to length.

In this work, we extend RAPID in several dimensions:

1. A signature might appear in all the faulty executions, and also in many or most of the correct ones[1]. On the other hand, another signature might appear in all the faulty executions, but never or rarely in the correct ones. These two signatures are different in terms of their discriminative power. While RAPID does not distinguish them, our method is able to inform that the second signature is more discriminative. To do this, we compare and contrast faulty and correct executions at the *signature* level in addition to the *event* level comparison performed by RAPID by using Tarantula.

2. It might be the case that a behavior is obeyed by all except one or a few faulty executions. RAPID does not categorize this as a signature. In our work, we would consider this as a signature if the behavior has enough discriminative power to distinguish between the faulty and correct traces. We thus tolerate minor imperfection and slight variation of signatures in the traces.

3. We use a graph-based representation rather than a sequence-based representation to achieve: (1) scalability, as we can compress long sequences to compact graphs with loops; and (2) expressiveness, as a graph can express more relationships than a sequence – a graph is able to specify both total and partial order, while a sequence is only able to specify total order.

4. Rather than sorting the returned signatures by length, we sort the returned signatures by their discriminative power. The algorithm is parameterized to return top-k graphs in the descending order of their discriminative scores.

[1]This is possible with Tarantula threshold set at 0.6 as used by RAPID.

2.2 Bug Localization

Recently, there have been a lot of studies on bug localization and automated debugging. These studies take as input two sets of execution traces and report candidate single-line locations where a bug potentially occurs. Bug localization studies include [12, 17, 24, 5, 14, 15]. Due to the space limitation, we only list some of these promising studies and describe them in brief.

Jones and Harrold proposed Tarantula in [12] which ranks a program statement based on its level of suspiciousness. Conceptually, a program statement is more suspicious if it appears in the faulty runs more frequently than in the correct runs. Given a faulty run and a set of correct runs, Renieris and Reiss presented a fault localization tool WHITHER [17] that compares a faulty execution to the nearest correct run and reports the most suspicious locations in the program. Zeller and Hildebrandt proposed a technique called Delta Debugging that localizes the minimum state change that results in a bug [24]. The technique swaps part of the memory state of the program under investigation and iteratively searches for the minimum swap or state change that results in a fault in a binary-search-like fashion. In a later work, Cleve and Zeller extended the work in [24], by incorporating a search for cause transitions, namely locations in the program where a state change on a variable stops becoming the cause for the bug (found using Delta Debugging) and the bug is now caused by a state change on another variable, in their tool called AskIgor [5]. Liblit *et al.* proposed a technique to search for predicates whose true evaluation correlates with failures [14]. While the work by Liblit *et al.* only considers if a predicate is ever evaluated as true, Liu *et al.* extended the work by incorporating information on the outcome of multiple predicate evaluations in a program run in their tool called SOBER [15].

In contrast to the above work that reports candidate single-line locations where a bug potentially occurs, we follow the strategy of bug signature identification, in which a bug is reported together with its context. The context could help a programmer to identify, understand and fix the bug. Furthermore, a graph can be thought of as a multi-dimensional discriminative feature that separates the faulty executions from the correct ones. Also, in contrast to the work in [17, 24, 5] we analyze the faulty and correct traces en masse since we would like to compute for discriminative bug signatures. Hence, we extend and complement the above past studies on bug localization.

Some other approaches use program slicing to provide for context [2, 6, 11, 25]. However often the slices are too large and include too much information which might potentially dilute important bug-related information among the noise.

Jiang and Su developed a technique to extract faulty control flow paths from program executions [11]. The technique first finds discriminative predicates, and groups these predicates according to their similarity across multiple executions. A path is then traced heuristically that tries to connect similar predicates belonging to the same group. A final filtering step is performed to remove redundant paths as well as paths involving an intra-procedural path. Similar to RAPID [7], since we analyze the program traces globally, the discriminative graphs extracted by our approach guarantee that they express *feasible paths* in the program that *actually run* during testing. On the other hand, the returned path by the above approach by Jiang and Su only returns *possible contexts* approximated heuristically. Also, in contrast to their work, we do not relate similar predicates, but rather mine combinations of predicates in the form of graphs that could further discriminate correct from faulty runs. Furthermore, the graphs are able to express contextual information with both total-order and partial-order relationships. A single predicate might not be discriminative, but an ordered and interacting set of predicates can be highly discriminative.

2.3 Graph Mining

Frequent subgraphs are the subgraphs that can be discovered from a collection of graphs with a frequency no less than a user-specified support threshold. Recent studies have developed a lot of efficient frequent subgraph mining methods through two major approaches: an Apriori-based approach and a pattern-growth approach. The Apriori-based approach [10, 13, 20] starts with graphs of small size and proceeds in a bottom-up manner. At each iteration, the size of newly discovered frequent subgraphs is increased by one. On the other hand, the pattern-growth approach [23, 3, 8, 16] extends a frequent graph by adding a new edge in every possible position. A key difference between our discriminative graph mining method and these mining algorithms is that, our method has two sets of graphs as input with labels of *correct* and *faulty*. Then our method directly searches for the most discriminative subgraphs which distinguish faulty executions from correct ones. On the other hand, these mining algorithms only have a single set of graphs without labels as input. Given a user-specified minimum frequency threshold, they will generate the whole set of frequent subgraphs whose frequency in the input graph dataset are above the threshold. However, most of the discovered frequent subgraphs may not be discriminative in terms of distinguishing correct and faulty runs, thus are not bug relevant. Furthermore, the graph mining result set could be very large, making it very hard for end users to digest and use.

Ting and Bailey [19] proposed to mine minimal contrast subgraphs which appear in the positive graphs but never in the negative ones. A recent study by Christodorescu *et al.* [4] uses the minimal contrast subgraph mining algorithm for mining specifications of malicious behaviors. The idea of identifying contrasts through graph mining is very close to ours. The major differences are the principles of the two mining algorithms: [19] considers only the contrast between positive and negative graphs, but ignores the commonality between multiple positive graphs, while our LEAP method considers both aspects. Therefore, the discriminative subgraphs generated by LEAP are representative of the faulty executions. In addition, the time complexity of [19] is quite high – the runtime increases exponentially with the number of input negative subgraphs (i.e., correct executions).

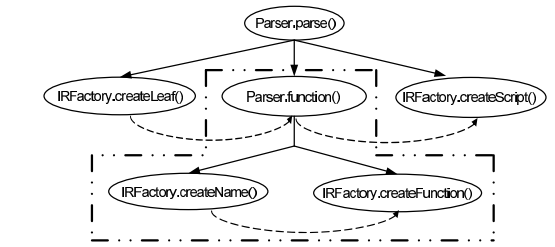
3. PRELIMINARY CONCEPTS

Software can be traced at different levels of granularity: method, basic block and statement. We consider two different levels of granularity, namely, method and basic block. After an instrumented program is run, a long series of events corresponding to method or basic block, depending on the level of granularity, is generated. These long sequential traces can then be coiled to form software behavior graphs.

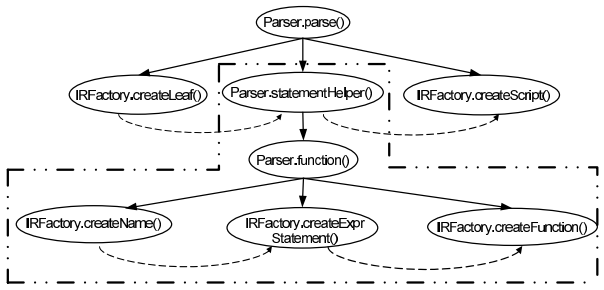
A method level behavior graph $G(\alpha_m)$ is a directed graph representing a method level program execution trace α_m. The vertex set denoted by $V(G(\alpha_m))$ includes all the methods appearing in α_m. The set of edges in $G(\alpha_m)$ is denoted by $E(G(\alpha_m))$ and corresponds to a set of vertex pairs. Each pair (v_i, v_j) corresponds to an edge between nodes v_i and v_j in $G(\alpha_m)$. There are two types of edge labels, namely: *call* and *trans*. An edge $(v_i, v_j) \in E(G_t(\alpha_m))$ is labeled as *call* if and only if method j is called by method i in α_m. Similarly, an edge $(v_i, v_j) \in E(G_t(\alpha_m))$ is labeled as *trans* if and only if method j is called right after method i returns, with no method calls being made between the two method invocations in α_m. The *trans* edges in the method level graphs capture relationships among sibling methods called consecutively. We consider these two types of edges as they capture two different relationships among method calls and enrich the expressiveness of both the input graphs and the mined signatures. These in turn would enable better differentiation of faulty and correct behaviors.

Similarly, a basic block level behavior graph $G(\alpha_b)$ is a directed graph representing a basic block program execution trace α_b. The vertex set includes all basic blocks appearing in α_b. There are three types of edge labels, namely: *call*, *trans* and *return*. An edge $(v_i, v_j) \in E(G(\alpha_b))$ is labeled as *call* if and only if it corresponds to a method invocation, where basic block j is called by basic block i in α_b. Similarly, an edge $(v_i, v_j) \in E(G(\alpha_b))$ is labeled as *trans* if and only if basic block j is executed right after basic block i in α_b. Furthermore, an edge $(v_i, v_j) \in E(G(\alpha_b))$ is labeled as *return* if and only if it corresponds to method return, where basic block i returns to basic block j in α_b. The three edge types capture the different control flow relationships between the basic blocks in the program.

A software behavior graph G is a subgraph of another graph G' if there exists a subgraph isomorphism from G to G', denoted by $G \subseteq G'$. G' is called a super-graph of G.



(a) Partial Behavior Graph for a Correct Execution



(b) Partial Behavior Graph for an Erroneous Execution

Figure 1: Software Behavior Graphs

We call a subgraph in a software execution a ***partial software behavior graph*** which represents a subset of functions or basic blocks, and their relationships, i.e., method call, method return or transition.

Definition 1 (Subgraph Isomorphism). *For two software behavior graphs G and G', a* subgraph isomorphism *is an injective function $f : V(G) \rightarrow V(G')$, s.t., (1), $\forall v \in V(G), l(v) = l'(f(v))$; and (2), $\forall (u,v) \in E(G), (f(u), f(v)) \in E(G')$ and $l(u,v) = l'(f(u), f(v))$, where l and l' are the labeling functions of G and G', respectively. f is called an embedding of G in G'.*

Definition 2 (Frequency). *Given a software behavior graph dataset $D = \{G_1, G_2, \ldots, G_n\}$ and a subgraph g, the supporting graph set of g is $D_g = \{G | g \subseteq G, G \in D\}$. The frequency of g is $\frac{|D_g|}{|D|}$.*

Example 1. *Figure 1 shows the partial software behavior graphs from two different runs of the program Mozilla Rhino with a bug of number 194364. Mozilla Rhino is an open-source implementation of JavaScript written in Java [1]. The bug causes Rhino to process function-expression-statement differently from the intended behavior observed by Microsoft Internet Explorer (MSIE) and SpiderMonkey (SM) – Mozilla's C implementation of JavaScript. In the graph, each node represents a method, solid edges represent method calls and dashed ones for transitions. It is very clear that the faulty execution has a very different subgraph structure from the correct execution, as highlighted with the bounding boxes in Figure 1.*

Given a program, different inputs will result in different software behavior graphs and different outputs. Assume that we have two sets of faulty and correct traces. In the faulty set, some bugs are manifested for each trace; while in the correct set, there is no appearance of any bug. We can then generate a set of software behavior graphs of correct executions and faulty executions respectively, for localizing bugs and analyzing their contexts with a graph mining approach.

4. MINING BUG SIGNATURES

Our bug signature mining framework is illustrated in Figure 2. Given an instrumented program and a set of test cases, running the program with the test cases will produce a set of execution traces. These execution traces are then converted to form behavior graphs either at a method level or basic block level depending on the level of granularity considered. A behavior graph coils a potentially very long sequence of events into a compact representation.

These graphs are then further filtered to remove edges which are non-suspicious. An edge corresponds to either a method call, method return or transition (either at a basic block or method level). An edge connects two nodes, each corresponding to a method (or a basic block).

Given a set of behavior graphs, two edges are considered to be equivalent if they correspond to the same type of relationship between the same pair of two methods or basic blocks. Formally, given two graphs $G1 = (V1, E1)$ and $G2 = (V2, E2)$, two edges $e1 = (v1, v2) \in E1$ and $e2 = (u1, u2) \in E2$ are the same or equivalent iff:

1. Vertices $v1$ and $u1$ correspond to the same method (or basic block), i.e., $l(v1) = l'(u1)$, where l and l' are labeling functions of $G1$ and $G2$ respectively.

2. Vertices $v2$ and $u2$ correspond to the same method (or basic block), i.e., $l(v2) = l'(u2)$.

3. Edges $e1$ and $e2$ have the same label, i.e., $l(e1) = l'(e2)$.

To identify whether an edge edg is suspicious, we evaluate the following parameter-free predicate $susp_{edg}$, which is evaluated to be either true or false:

$$susp_{edg} = \frac{failed(edg)}{passed(edg)} > \frac{totalfailed}{totalpassed}$$

The operators $failed(edg)$ and $passed(edg)$ report the number of faulty program behavior graphs that include the edge edg and the number of correct program traces that include the edge edg respectively. Only suspicious edges would be retained. Any nodes that have no edges would also be removed from the resultant pre-processed graphs.

The above approach is similar to Tarantula [12] with notable differences. Rather than filtering statements corresponding to nodes in the graphs, we filter edges corresponding to relationships between basic blocks or methods in the input program traces. Also, different from the filtering step of RAPID which uses Tarantula, our filtering approach is parameter free. Hence, the result is not dependant on the correct setting of the parameter, which can sometimes be hard to do.

We illustrate the power of discriminative graphs in aiding debugging by means of the following example. The code snippet shown in Table 1 shows a simple buggy method in C++ which tries to replace the first occurrence of either sx or sy with sz in a string array arr of length len.

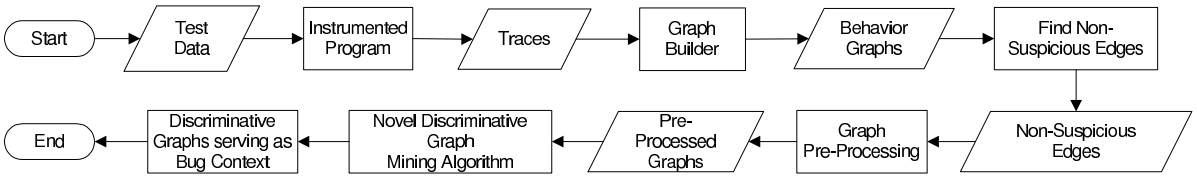


Figure 2: Context-Based Bug Localization Framework

```
1: void replaceFirstOccurence (string arr [], int len, string sx, string sy, string sz) {
2:    for (int i=0;i<len;i++) {
3:        if (arr[i]==sx){
4:            arr[i] = sz;
5:            // a bug, should be a break;
6:        }
7:        if (arr[i]==sy)){
8:            arr[i] = sz;
9:            // a bug, should be a break;
10:       }
11:   }
12: }
```

Table 1: Buggy Code Snippet – An Example

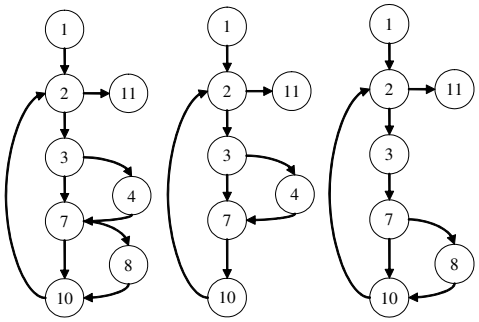


Figure 3: Control Flow Graph (CFG) of Code in Table 1 and Behavior Graph of Traces 3 & 4 (left), Behavior Graph of Trace 1 (middle), and Behavior Graph of Trace 2 (right). All edges in the behavior graphs are labeled as trans.

The code contains several bugs. Rather than replacing the first occurrence of either sx or sy with sz, it replaces *all* occurrences of sx and sy with sz. Consider the following set of test cases.

No	arr	sx	sy	sz
1	{a, b}	a	g	1
2	{a, b}	g	a	1
3	{a, g}	a	g	1
4	{a, g}	g	a	1

The first two test cases result in correct execution traces, while the third and fourth result in failures. The Control Flow Graph (CFG) of the program is shown in Figure 3. Each number in the CFG corresponds to the line number where the start of the corresponding basic block is located. Running the above test cases on the instrumented version of the code in Table 1, produces a set of execution traces shown below (each corresponds to a list of ids of the basic blocks traversed during the respective execution):

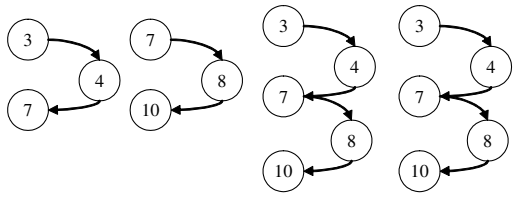


Figure 4: Pre-processed graphs for the four execution traces. All edges are labeled as trans.

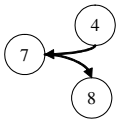


Figure 5: The discriminative subgraph. All edges are labeled as trans.

No	Trace
1	⟨1, 2, 3, 4, 7, 10, 2, 3, 7, 10⟩
2	⟨1, 2, 3, 7, 10, 2, 3, 7, 8, 10⟩
3	⟨1, 2, 3, 4, 7, 10, 2, 3, 7, 8, 10⟩
4	⟨1, 2, 3, 7, 8, 10, 2, 3, 4, 7, 10⟩

Only four edges would be suspicious and be retained in the pre-processed graphs. The resultant pre-processed graphs generated are shown in Figure 4.

A discriminative graph is shown in Figure 5. It highlights that the problem occurs when the basic block starting at line 4 is executed with that of line 8. Note that the bug signature is not minable by RAPID. This is the case as the bug appears with slightly different signatures at traces 3 and 4, i.e., at trace 3, basic block 4 is executed before 8, while at trace 4, basic block 8 is executed before 4.

5. MINING DISCRIMINATIVE GRAPHS WITH STRUCTURAL LEAP SEARCH

Traditional graph mining algorithms [10, 13, 20, 23, 3, 16] mine all frequent subgraphs from a graph dataset with-

out labels. With frequency as the only mining constraint, those frequent subgraphs may not have the discriminative power to differentiate faulty executions from correct ones. Hence, those traditional graph mining algorithms cannot be used directly to solve the bug signature mining problem. In this section, we propose a novel graph mining algorithm to directly extract discriminative subgraphs for bug signature identification. We will first formulate the graph mining problem as an optimization problem for mining the most discriminative subgraph. We then extend the mining algorithm to mine top-k discriminative subgraphs for a ranked list of potential bug locations.

5.1 Mining The Most Discriminative Subgraph

Given two sets of software behavior graphs from correct and faulty executions, we aim to find some partial behavior graphs which are highly indicative of bugs. Intuitively, a partial software behavior graph is related to a bug with high probability if it appears frequently in the set of faulty executions, but rarely in the set of correct ones. Theoretically, we can design an objective function $F(g)$ to evaluate the significance of a subgraph g as an indication of a bug. Then our goal becomes finding the optimal subgraph from a set of correct and faulty runs wrt. the objective function F. Formally, the problem is defined as:

Given a set of graphs with class labels, $D = \{G(\alpha_i), y_i\}_{i=1}^n$, where $G(\alpha_i) \in \mathcal{G}$ is a software behavior graph representing an execution and $y_i \in \{\pm 1\}$ is the class label representing a correct or faulty status, an objective function F, find a subgraph g^* such that $g^* = \text{argmax}_g F(g)$.

In data mining and machine learning, discriminative measures such as information gain, cross entropy and Fisher score are popularly used to evaluate the capacity of a feature in distinguishing instances from different classes. In this work, we use information gain as the objective function. According to information gain, if the frequency difference of a subgraph in the faulty executions and the correct executions increases, the subgraph becomes more discriminative. A subgraph which occurs frequently in faulty executions but rarely in correct executions will have a very large information gain score, and indicate that the corresponding partial software behavior graph is very discriminative to differentiate faulty executions from correct ones. Such discriminative graph highlights the structural contrast between faulty and correct graphs. If we use c to denote the class labels of correct or faulty runs, and use g to represent a subgraph, then information gain of g is defined as in Eq.(1).

$$IG(c|g) = H(c) - H(c|g) \tag{1}$$

where $H(c) = -\sum_{c_i \in \{0,1\}} p(c_i) \log p(c_i)$ is the entropy and $H(c|g) = -\sum p(g) \sum_{c_i \in \{0,1\}} p(c_i|g) \log p(c_i|g)$ is the conditional entropy given the subgraph g.

To find the subgraph with the highest information gain, a naive solution could be: enumerate all possible subgraphs from the graph dataset $D = \{G(\alpha_i), y_i\}_{i=1}^n$, then rank subgraphs according to their objective function scores and select the one with the highest score. However, this method is not scalable due to the following two reasons: (1) enumerating all possible subgraphs generates an exponential number of graph patterns, due to the combinatorial explosion between graph vertices and edges; and (2) there is a lot of redundancy in the mining result set, due to the subgraph-supergraph relationship. If a subgraph is very discriminative, its supergraph with an extra edge is likely to be discriminative as well. However, they may actually pinpoint the same bug. Hence, it is unnecessary and infeasible to thoroughly mine the whole set of subgraphs.

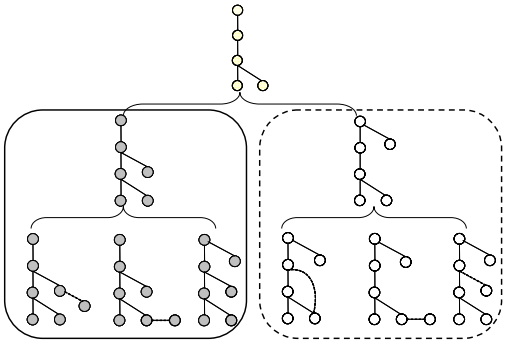


Figure 6: Structural Similarity in a Graph Search Tree

A more efficient solution is to directly mine the most discriminative subgraph g^* from D. In this scenario, the objective function of information gain needs to be integrated into the graph mining process. When a subgraph is generated, the information gain score is calculated. As we traverse the search space, we always keep track of the most discriminative subgraph seen so far. The globally optimal subgraph is returned when the search space is thoroughly explored.

Based on this idea, we adopt a recently proposed graph mining algorithm, LEAP [22], to directly mine the most discriminative subgraph from behavior graphs of correct and faulty executions. To speedup the mining process and prune unpromising search spaces, [22] proposed a novel concept, *structural leap search* and integrated it into LEAP. This is based on the observation that, sibling branches in the graph pattern search tree exhibit strong similarity not only in pattern composition, but also in their embeddings in the graph datasets, thus having similar frequency distributions and objective function scores. Figure 6 illustrates the structural similarity idea through an example pattern search space. The left branch and right branch have a very high structural similarity since they are generated from the same ancestor pattern. If the similarity between these two sibling branches exceeds a certain threshold, LEAP can skip the right branch after it has traversed the left sibling branch, since the subgraphs and their objective function scores between the sibling branches are likely similar. Structural leap search will focus on searching distinct branches and reduce the need for thoroughly searching similar branches in the pattern search tree. It limits the chance of missing the most discriminative subgraph, and finally generates a (near)-optimal [2] subgraph in the search space. Although in the worst case, the number of candidate subgraphs examined by LEAP is still exponential in terms of the size of an input graph, empirical evaluation demonstrated that it is very efficient and scalable. Experimental study in [22] showed that LEAP is up to an order of magnitude faster than the popularly used branch-and-bound search strategy in graph mining.

[2] LEAP can be parameterized to mine either a near-optimal subgraph or the optimal subgraph with a slight overhead.

Algorithm 1 Mining Top-K Discriminative Graph Patterns
Top-K LEAP(D, F, k)

Input: Graph dataset D, objective function F, k
Output: Top-k discriminative subgraphs G_k.

```
1: G_k = ∅;
2: for i from 1 to k
3:       g_i^* = LEAP(F, G_k);
4:       if g_i^* = ∅
5:           break;
6:       G_k = G_k ∪ {g_i^*};
7: end for
8: return G_k;
```

5.2 Mining Top-K Discriminative Subgraphs

In the bug localization problem, it does not suffice to report the single best location only. Rather, it would be more informative to generate a ranked list of discriminative subgraphs with descending scores which are highly indicative of bugs and their contexts. To handle such requirements, we propose an extension from LEAP to mine top-k discriminative subgraphs. Formally, the problem is defined as:

Given a set of software behavior graphs with class labels, $D = \{G(\alpha_i), y_i\}_{i=1}^{n}$, an objective function F, find k subgraphs $G_k = \{g_i^*\}_{i=1}^{k}$ from D which maximize $\sum_{i=1}^{k} F(g_i^*)$.

Accordingly, the LEAP algorithm will be modified to generate k discriminative subgraphs through k iterations. Initially LEAP finds the most discriminative subgraph g_1^* from D wrt. the objective function F and inserts it into G_k. It iteratively finds a subgraph g_i^* with the highest F function score which is also different from all existing subgraphs in G_k. Such a subgraph g_i^* is inserted into G_k and this process is iterated for k times. Finally, G_k is returned. If there exist less than k subgraphs from D, the search process will terminate early. The extended algorithm is called Top-K LEAP, as shown in Algorithm 1.

6. PRELIMINARY EXPERIMENTS

In this section, we describe our experiments to test the performance of our bug signature identification approach. We experiment with the Siemens benchmark datasets. These datasets were developed by researchers of Siemens Corporation to test the adequacy of test coverage strategies [9]. These datasets are based on a set of seven programs which are seeded by commonly found bugs. Each version is seeded with one unique bug. These datasets have been used by various research work on bug localization, e.g., [5, 14, 15].

We compare and contrast our technique with RAPID, the state-of-the-art bug signature identification tool by Hsu *et al.* [7]. In [7], the utility and power of the bug signature identification approach have been illustrated by means of an example. In this paper, we do more evaluation by analyzing the entire Siemens dataset under objective quantitative performance metrics. In sub-section 6.1, we describe the performance metrics used and explain why they are useful. We compare these metrics with the distance-based metric used by studies that return single-line candidate bug locations without the context, e.g., [5, 15] and explain the differences.

	RAPID			Top-K LEAP		
Prog.	**Pre.**	**Rec.**	**Size**	**Pre.**	**Rec.**	**Size**
tcas	82.9	82.9	8.0	85.9	95.1	5.0
ptok	71.4	71.4	4.0	85.7	100	4.3
ptok2	20.0	20.0	2.7	36.0	60.0	2.9
sched	33.3	33.3	2.3	54.1	66.7	3.6
sched2	0.0	0.0	N/A	24.2	30.0	2.2
tinfo	21.7	21.7	2.5	69.6	78.3	2.4
rep	53.1	53.1	5.1	54.4	81.3	2.9
Avg.	40.4	40.4	4.1	58.5	73.0	3.3

Table 2: Result - Method Level

	RAPID			Top-K LEAP		
Prog.	**Pre.**	**Rec.**	**Size**	**Pre.**	**Rec.**	**Size**
tcas	90.2	90.2	11.3	88.3	100	3.8
ptok	100	100	9.7	85.7	100	4.8
ptok2	65.0	70.0	7.2	74.0	100	3.4
sched	75.0	77.8	5.1	86.7	88.9	3.2
sched2	40.0	40.0	2.6	52.0	80.0	2.8
tinfo	56.5	56.5	15.4	55.0	87.0	3.6
rep	80.5	81.3	20.7	78.1	81.3	4.9
Avg.	72.5	73.7	10.3	74.3	91.0	3.8

Table 3: Result - Basic Block Level

6.1 Performance Metrics

The performance of our algorithm is evaluated in terms of two commonly used measures of *precision* and *recall*. Precision refers to the proportion of returned results that highlight the bug. Recall refers to the proportion of bugs that can be discovered by the returned bug signatures: be it a set of sequential patterns (as returned by RAPID [7]) or a set of discriminative graphs (as returned by Top-K LEAP).

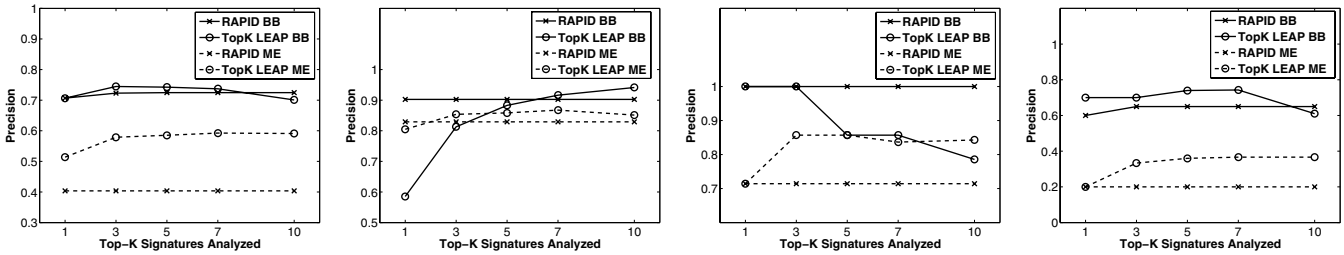
To evaluate precision and recall, the fourth author manually browsed through the returned bug signatures and marked whether each of the signatures (or contexts) is correct or not. The measures of precision and recall can then be computed.

6.2 Experimental Results

The following are the results for precision and recall for the seven Siemens datasets. For each dataset we take the average of precision and recall across all the buggy versions. We compare the precision and recall of our approach with those of RAPID. We consider the top 5 signatures among the sorted results returned by RAPID and our approach Top-K LEAP. We believe a bug-signature identification tool is only practical if only a small set of results need to be analyzed. We consider two types of datasets based on tracing at the method and basic block levels. The results for the method level for both RAPID and Top-K LEAP are shown in Table 2[3]. The corresponding results for the basic block level are shown in Table 3. The columns *Pre.*, *Rec.* and *Size* correspond to the average precision, recall and size (in terms of the number of nodes) of the returned signatures.

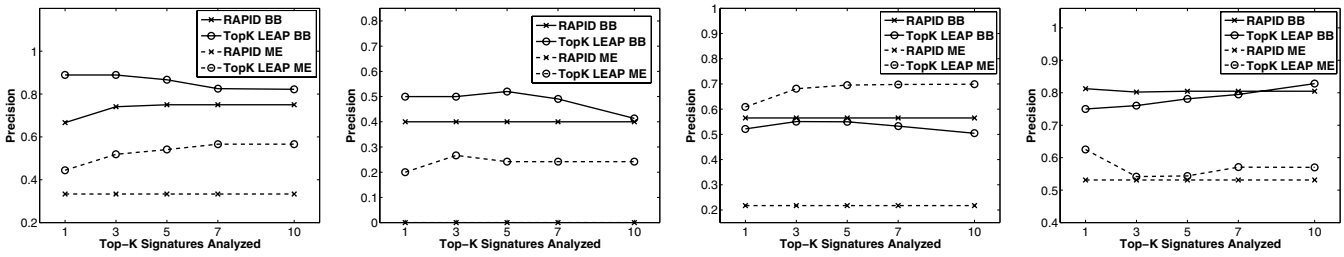
For the method level, the above results show that Top-K LEAP has on average 18.1% higher precision and 32.6% higher recall than RAPID. For the basic block level, Top-K LEAP has on average 1.8% higher precision and 17.3% higher recall than RAPID. Among the datasets, we find that schedule2 (sched2) is the hardest one as the precision and

[3]We abbreviate the names of some programs: print_tokens $\mapsto$ ptok, print_tokens2 $\mapsto$ ptok2, schedule $\mapsto$ sched, schedule2 $\mapsto$ sched2, tot_info $\mapsto$ tinfo, and replace $\mapsto$ rep.



(a) Average Precision (b) tcas (c) print_tokens (d) print_tokens2

Figure 7: Precision Plots I



(a) schedule (b) schedule2 (c) tot_info (d) replace

Figure 8: Precision Plots II

recall measures for both Top-K LEAP and RAPID are the lowest at both the method and basic block levels.

We further examined the size of the returned bug signatures. The average size of discriminative subgraphs returned by Top-K LEAP over all datasets is smaller than 4 (nodes) for both the method and basic block levels. In contrast, the size of bug signatures returned by RAPID is larger. In addition, the average size increases significantly at the basic block level for RAPID.

To further analyze the results, we plot the results, i.e., precision and recall for Top-K LEAP and RAPID at both the method (abbr. as **ME**) and basic block (abbr. as **BB**) levels, with respect to the top-k results returned, where k varies from 1 to 10. We plot 5 data points corresponding to $k = 1$, 3, 5, 7 and 10. These graphs are shown in Figures 7–10. At times increasing the number of signatures analyzed reduces the precision. However, we find that this is acceptable, as a programmer is likely to stop searching for the bug once he/she finds a signature that captures the bug.

From the experiments, we find that Top-K LEAP is able to complete subgraph mining on every dataset between less than a second to a maximum of 258 seconds. On the other hand, we find that BIDE which is used by RAPID is not able to complete successfully for version 6 of the replace dataset (at the basic block level). It cannot complete mining even after we left it to run for more than 10 hours[4]. A major reason for this runtime difference is the compactness of data representation. Sequential traces are potentially very long. This significantly slows down the sequential pattern mining process even when the support threshold is set to 100%[5]. The mining process runs with potentially exponentially increasing cost as the length of the traces increases. On the other hand, our graph based approach is able to "coil" the traces to form a compact representation in the form of behavior graphs.

6.3 Experience

This section describes the experience we have with the tool on the Siemens dataset. In particular, we highlight several graphs that we mine from the Siemens dataset to show the power of bug signature or contextual information via discriminative graph mining.

In version 7 of program `schedule`, methods `upgrade_process_prio` and `unblock_process` contain two variants of a bug shown below:

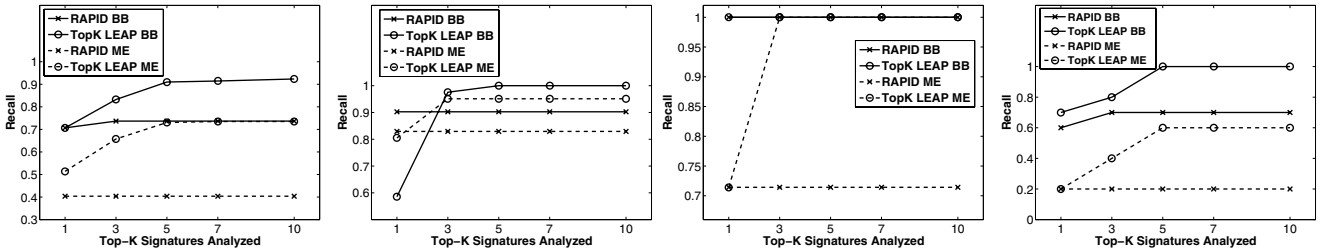
```
1 upgrade_process_prio(prio, ratio){
      ...
2     n = (int) (count*ratio+1);
3     if(ratio == 1.0) n--; //added code
4     proc = find_nth(src_queue, n);
...}

5 unblock_process(prio, ratio){
      ...
6     n = (int) (count*ratio +1);
7     if(ratio == 1.0) n--; //added code
8     proc = find_nth(src_queue, n);
...}
```

Lines 3 and 7 of the buggy version are mistakenly added and correspond to the bugs. When either one of the two `n--;` statements is executed, the variable `n` will be mistakenly decreased by 1; the method `find_nth` will return a wrong value and the program may produce a wrong output. Our technique is able to find the two bugs, while RAPID can not. The reason lies on the fact that neither of the two statements (lines 3 and 7) happens in all of the faulty executions. A faulty trace can only contain either line 3 or 7. Line 3 appears in almost the same number of faulty traces as line 7. Since RAPID only captures longest common subsequence of the faulty traces, it is not able to capture the bug. In contrast, our discriminative pattern mining technique is able to

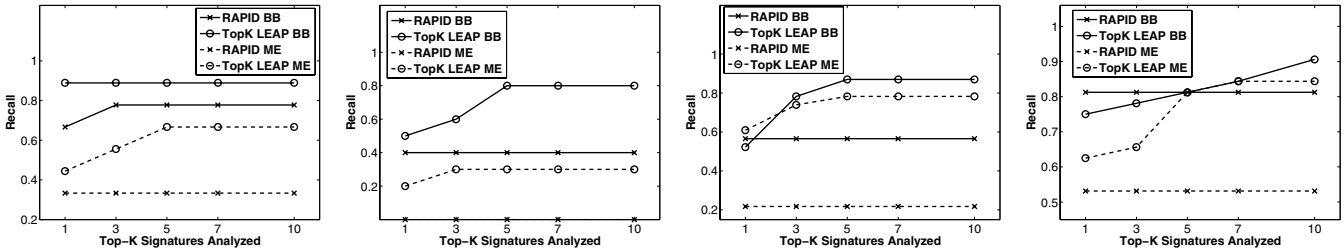
[4]Similarly, for version 10 of the print_tokens2 dataset (at the basic block level), it runs for a few hours before throwing an out of memory exception.

[5]As RAPID uses BIDE to mine for Longest Common Subsequences (LCSs), a support threshold of 100% is used. At a lower support threshold, the performance is worse. When the support level is set at 70%, 6 out of the first 10 versions of the replace dataset (at the basic block level) are not able to run to completion even after running for an hour.



(a) Average Recall (b) tcas (c) print_tokens (d) print_tokens2

Figure 9: Recall Plots I



(a) schedule (b) schedule2 (c) tot_info (d) replace

Figure 10: Recall Plots II

capture the bug via two signature variants capturing lines 3 and 7. This is the case as we capture for patterns that appear more frequently in the faulty traces than the correct traces. Each of these discriminative patterns can appear in all, in most or in a few of the faulty traces. This allows our technique to tolerate minor imperfection and variants of a bug signature as verified by the above example.

Consider another example from version 18 of `tot_info` program. The code snippet capturing the bug is shown below:

```
1 InfoTbl( r, c, f, pdf ){
2     rdf = r-1; cdf = c-1; //->basic block 1
3     if ( rdf == 0 || cdf == 0 ) { //bug
4         info = -3.0; //->basic block 2
5         goto ret3;
      }
6     N = 0.0; //->basic block 3
7     for ( i = 0; i < r; ++i ){
8          //->basic block 4,
9     }
10    //->basic block 5
11    if ( N <= 0.0 ){...}
12    ...
13 }
```

In the correct version of `tot_info` program, line 3 of the above code snippet checks whether any of the parameters (`rdf` and `cdf`), corresponding to the dimensions of the `InfoTbl` table, is zero or a negative number. If this is the case, an error message needs to eventually be displayed (by returning -3 to the main function).

Line 3 is mistakenly changed from `if(rdf<=0||cdf<=0)` to the one shown in the code snippet above. Thus, when `rdf` or `cdf` is less than zero (i.e., `r` or `c` is less than or equal to zero), the control flow which should have gone to line 4 will go to line 6 (marked as basic block 3).

For this bug, the sequence based technique does not find anything, but our technique gets a context graph with a *trans* edge that links `basic block 3` and `basic block 5`, which represents that `basic block 5` is executed immediately after `basic block 3`. The condition that basic block 5 is executed right after 3 only happens in the faulty execution but never in the correct execution. This happens when `r` <= 0, the condition when the bug is manifested. The programmer could then use this information to find the bug, e.g., by investigating locations in the method where the parameter r and its derivatives (e.g., rdf) are involved.

Note that basic block 1 is executed by all calls to method InfoTbl, hence it is not discriminating. Execution of basic block 3 or basic block 5 alone does not reveal the fault. Rather when they are executed together, the fault is revealed. This is a path-dependent fault. Existing fault localization techniques that report individual candidate buggy statements are less likely to be effective. Even if they report lines 6 and 10 respectively, they are not able to relate these lines together to form the context of the bug.

RAPID can not find the bug signature because none of the basic blocks related to the bug is marked as suspicious. The preprocessing stage of RAPID only marks one basic block not related to the bug as being suspicious in this case. Since codes corresponding to `basic block 3` and `basic block 5` are executed with a similar relative frequency in both correct and faulty runs, these two blocks are not marked as being suspicious by RAPID, which utilizes Tarantula at a relatively low filtering threshold of 60%.

Our graph-based technique captures edges, which represent relations between basic blocks. We are then able to find this suspicious relation between non-suspicious basic blocks. Hence, we could then capture the corresponding path-dependent bug that RAPID is not able to capture.

6.4 Threats to Validity

Similar to other empirical studies, there are several threats to validity in interpreting the results. First, we employ a manual process to identify good and bad bug signatures. The manual labeling process might be prone to errors. Second, we experimented with relatively small-scale C programs in the Siemens test suite. The results might not be translatable to larger programs. We have not experimented with programs written in other programming languages either.

However since the signatures we mine make use of discriminating control flow information and the concept of control flow is generic, the result is likely to apply to other types of programs.

7. DISCUSSION & FUTURE WORK

In this work, we only consider the control flow information when building graphs and extracting discriminative signatures. We try to make the signature rich enough by considering various edge types so that it has enough expressiveness to discriminate the buggy cases. We find that at times this is not sufficient as the data flow information matters. The same control flow could be executed by both buggy and normal behaviors, which might differ only on the data that flows in along this control flow. In the future, we are looking into building richer graphs that incorporate both control and data flows by incorporating predicates into the graphs. A new mining strategy might be required.

To limit the search space in graph mining, we only mine connected subgraphs. The connectivity constraint together with the removal of non-suspicious edges may cause us to miss some signatures. In the example shown in Table 1 we miss the signature that links node 8 to node 4 (only a signature that links node 4 to node 8 is returned – the signature is directional), as nodes 8 and 4 are separated by a series of edges some of which are non-suspicious. In the future, we are looking for methods to address this limitation.

So far we only use a manual labeling process to evaluate mined graphs as good or bad. In the future, we plan to develop a metric to measure how good or bad a returned graph is. This measure could be defined as the graph edit distance to the ideal bug signature.

We have described some interesting context-related bugs that we have discovered. In the future, we plan to conduct a larger-scale case study and/or a user-experience study to investigate how effective the context-based bug signatures in helping programmers to detect and fix bugs. We would also like to investigate the utility of our approach in debugging multi-threaded applications.

8. CONCLUSIONS

In this work, we propose a new bug signature identification technique based on top-k discriminative graph mining. We extend RAPID, the pioneer work on bug signature identification by Hsu *et al.* in the following dimensions: (1) we propose a graph-based representation which is more compact and scalable in representing long traces; (2) we mine for graph patterns which are able to express contextual information incorporating both partial and total ordering of events; (3) we compare and contrast faulty and correct traces at both event and pattern levels for producing a set of multi-dimensional discriminative features; and (4) we allow and account for imperfections in traces and slight variations of bug patterns. Our work first produces two sets of graphs corresponding to the faulty and correct traces. These graphs are then pre-processed to filter off non-suspicious edges. A top-k discriminative graph mining algorithm is then run to produce a list of candidate discriminative graphs that serve as bug signatures identifying both the location and the context of a bug. We perform a set of experiments based on the Siemens benchmark dataset. Experimental results show that our technique achieves up to 18.1% higher precision and 32.6% higher recall than RAPID.

Acknowledgement We would like to thank H. Hsu and A. Orso for their advice on the comparison of their work [7] with the work by Jiang and Su in [11]. We would like to thank the reviewers for their valuable comments and suggestions.

9. REFERENCES

[1] Rhino - JavaScript for Java. www.mozilla.org/rhino.
[2] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE*, 1995.
[3] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, 2002.
[4] M. Christodorescu, S. Jha, and C. Kruegel. Mining specification of malicious behavior. In *ESEC/FSE*, 2007.
[5] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.
[6] R. DeMillo, H. Pan, and E. Spafford. Critical slicing for software fault localization. In *ISSTA*, 1996.
[7] H. Hsu, J. A. Jones, and A. Orso. RAPID: Identifying bug signatures to support debugging activities. In *ASE*, 2008.
[8] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *ICDM*, 2003.
[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, 1994.
[10] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, 1998.
[11] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, 2007.
[12] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.
[13] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, 2001.
[14] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.
[15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *ESEC/FSE*, 2005.
[16] S. Nijssen and J. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, 2004.
[17] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.
[18] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.
[19] R. M. H. Ting and J. Bailey. Mining minimal contrast subgraph patterns. In *SDM*, 2006.
[20] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *ICDM*, 2002.
[21] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE*, 2004.
[22] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by scalable leap search. In *SIGMOD*, 2008.
[23] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, 2002.
[24] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. on Software Engineering*, 2002.
[25] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, 2006.