2011 IEEE 13th International Symposium on High-Assurance Systems Engineering

Bug Signature Minimization and Fusion

David Lo[1], Hong Cheng[2], and Xiaoyin Wang[3]
[1]School of Information Systems, Singapore Management University
[2]Dept. of Systems Engineering and Engineering Management, Chinese University of Hong Kong
[3]Institute of Software, EECS, Peking University
Email: davidlo@smu.edu.sg,hcheng@se.cuhk.edu.hk,wangxy06@sei.pku.edu.cn

***Abstract*—Debugging is a time-consuming activity. To help in debugging, many approaches have been proposed to pinpoint the location of errors given labeled failures and correct executions. While such approaches have been shown to be accurate, at times the location alone is not sufficient in helping programmers understand why the bug happens and how to fix it. Furthermore, a single location might not be powerful enough to discriminate failures from correct executions. To address the above challenges, there have been recent studies on extracting bug signatures which are composed of multiple locations appearing together in a particular order signifying an occurrence of a bug.**

The latest study on bug signatures by Cheng et al. models program executions as graphs. Two sets of graphs corresponding to failures and correct executions are then contrasted to extract the most discriminative connected subgraphs serving as bug signatures. However, there are two limitations: (1) returned signatures might not be minimal and (2) they can only capture localized bug context. In this work, we develop a signature minimization technique to capture minimal discriminative signatures. Also, we propose a technique of signature fusion to fuse disconnected subgraphs so that our method can capture bug contexts spanning multiple locations. Experimental study on Siemens and Space dataset shows the effectiveness of the proposed bug signature minimization and fusion techniques. Comparing with the state-of-the-art bug signature mining technique, we reduce the number of bugs missed by up to 57.7%, and reduce the average number of nodes traversed by up to 85.6%.

I. Introduction

Bugs are prevalent in day-to-day software development. As software is getting more complex, more complex bugs arise, causing the task of developing software systems even more time consuming. Many valuable resources that could have been spent in developing new systems or additional features are spent in fixing bugs. Not surprisingly, in the United States alone, 59.5 billion US dollars are spent annually on issues related to software bugs [13].

One key challenge in finding and fixing bugs is to localize the fault causing the failure. A failure could be manifested by either a crash in a system or a system output having a wrong value, or wrong data stored in the database, etc. Often, the location where a failure happens differs very much from the location of the fault. Developers typically need to backtrack from the failure point and insert multiple print statements to eventually localize the fault.

One research direction which aims to make debugging effort easier is to utilize automated techniques to trace from failures to faults. Namely, given a set of test cases causing the program to fail, and another set causing the program to run successfully, infer the location of the fault. These test cases could be run to collect software behaviors, which are the execution paths taken when a software is executed. Based on the positive behaviors and the negative ones (i.e., failures), the task is to infer the location in the program where the bug occurs. This task has been termed as *fault localization* which has been an active thread of study for many years [8], [12], [16], [3], [9], [10].

Most fault localization methods report single statements that are potential candidates for bugs, usually sorted according to some scoring function. While the approaches have been shown to be accurate, it is hard to understand why a bug occurs and how to fix the bug just by the flagged single statements alone. Also, at times the execution of a single statement is not powerful enough to differentiate correct from faulty executions. Sometimes two or more statements executed in a specific order might be the source of a failure.

To address the above issues there have recently been interests in finding bug signatures. Different from the single statements reported by bug localization work, bug signatures provide the contexts where the bugs occur. Bug signatures could be composed of multiple statements that occur together in a particular order whose occurrences signify a bug. The work was pioneered by Hsu et al. [6] and was improved by Cheng et al. [2] using their algorithm named Top-K LEAP.

The signatures returned by Top-K LEAP, though useful, are not guaranteed to be minimal in size. We have observed that the bug signatures may include extra unrelated program elements besides the true minimal signatures which are sufficient for revealing the bug and its context. Furthermore, the graph-based signatures are not able to capture bug contexts spanning two or more disjoint localities. This limitation is caused by the graph mining algorithm, as it only mines discriminative graphs that are connected. In this work, we further improve Top-K LEAP by Cheng et al. by proposing two novel techniques of signature minimization and fusion. Signature minimization aims at reducing the size of the reported signatures by removing unnecessary program elements, while signature fusion focuses on fusing two distant contexts if they, when combined, increase the suspiciousness of a signature.

We have performed experiments on Siemens datasets [7] and Space [4]. The experiments show that we reduce the number of bugs missed and the number of nodes traversed to find a *relevant* signature (i.e., a signature potentially helpful in debugging the error).

The contributions of this paper are as follows:

1530-2059/11 $26.00 © 2011 IEEE
DOI 10.1109/HASE.2011.36

1) We propose a new technique of signature minimization to generate smaller signatures for users to analyze.
2) We design a new technique of signature fusion that helps to connect disjoint signatures into a unified context with a higher discriminative power.
3) We show that our unified approach of signature minimization and fusion is able to reduce the number of bugs missed by up to 57.7% and reduce the number of program elements traversed by up to 85.6% on various Siemens datasets and a larger program Space with 9564 lines of code. We are the first to experiment bug signature identification on a larger non-Siemens dataset program.

The rest of the paper is organized as follows. Section II introduces related work. Section III defines the preliminary concepts. Section IV examines the problem of signature minimization, and proposes a complete solution as well as a greedy one. Section V formulates the bug signature fusion problem and proposes a solution. An integrated solution is presented in Section VI. Experimental evaluation is presented in Sections VII and VIII. Section IX concludes our study and describes some future work.

II. Related Work

Identifying Bug Signatures. RAPID is the pioneer work on bug signature identification. RAPID starts by accepting two sets of traces: faulty and correct. RAPID obtains traces containing statements corresponding to method entries and branch decisions. RAPID then uses Tarantula [8] to measure the score of suspiciousness of instrumented program elements and filters program elements with score less than 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 [14] with a frequency threshold of 100%. 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 returned.

Recently, Cheng et al. [2] proposed Top-K LEAP that mines bug signatures in the form of discriminative graphs. The method first coils both correct and faulty software executions into *software behavior graphs*. A software 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. Then Top-K LEAP extracts the most discriminative subgraphs which are highly indicative of bugs and their contexts from software behavior graphs of correct and faulty runs. The result has been shown to be better than RAPID due to several reasons including: a fully discriminative approach is used (discriminative comparison at signature level rather than event level), a graph-based approach is employed rather than a sequence-based one, etc.

In this work, we will build upon Top-K LEAP to mine *minimal bug signatures* and *bug signatures involving disconnected components*. To do this, we propose novel techniques of graph minimization and graph fusion. The results have shown that the proposed techniques could substantially help improving the quality of bug signatures.

Bug Localization. Jones and Harrold proposed Tarantula in [8] 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 [12] 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 [16]. Lucia et al. evaluated the effectiveness of many association measures for fault localization [11]. Wang et al. employed genetic algorithm for fault localization [15]. There have been many other studies on bug localization.

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.

III. Preliminary Concepts

This section describes a representation of software traces as software behavior graphs which is the input to our analysis. The concept of bug signature mining and the limitations of the state-of-the-art technique are also described.

A. Software Behavior Graph

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 sequence 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. We follow the software behavior graph representation in [2].

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 edge set denoted by $E(G(\alpha_m))$ includes a set of vertex pairs. Each pair (v_i, v_j) corresponds to an edge from v_i to v_j. There are two types of edges, namely, *call* and *trans*. An edge $(v_i, v_j) \in E(G(\alpha_m))$ is labeled as *call* if and only if method j is called by method i in α_m. Similarly, an edge $(v_i, v_j) \in E(G(\alpha_m))$ 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 should enable better differentiation of faulty and correct behaviors. We do not capture return edges, as they are redundant, i.e., if a method m1 calls m2, m2 should return to m1.

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*, where the call and trans edges are similarly defined as in the method level. 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.

B. Bug Signature Mining

Cheng *et al.* formulates the bug signature identification problem as a discriminative subgraph mining problem [2]. Their approach, called Top-K LEAP, returns the top-k discriminative bug signatures $G_k = \{g_1, ..., g_k\}$ according to a scoring function F.

Information gain [5], commonly used in data mining and machine learning as a discriminative measure, is chosen as the scoring function F. It 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. In bug signature identification problem, the class c_0 corresponds to correct executions and c_1 corresponds to faulty executions. The subgraph g corresponds to a bug signature. According to information gain, if the frequency difference of a subgraph/signature in the faulty executions and the correct executions increases, the subgraph/signature becomes more discriminative.

Definition 1 (Bug Signature Mining): Given a set of graphs with 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 $y_i \in \{Pass, Fail\}$ is the class label representing a correct or faulty status, an objective function F, find k subgraphs $G_k = \{g_i\}_{i=1}^{k}$ from D which maximize $\sum_{i=1}^{k} F(g_i)$.

The bug signature mining process in [2] is illustrated in Example 1.

Example 1: The code snippet shown in Table 1 shows a simple buggy method in C++ which takes in a list of unique characters in the input array unq of size len and tries to replace the first occurrence of either character cx or cy with cz. The code contains two bugs. Rather than replacing the first occurrence of either cx or cy with cz, it replaces all occurrences of cx and cy with cz. Consider the following set of test cases.

No	arr	cx	cy	cz
1	{a, b}	a	g	1
2	{a, b}	g	a	1
3	{a, g}	a	g	1
4	{a, g}	g	a	1

TABLE I
BUGGY CODE SNIPPET: AN EXAMPLE

```
1: void replaceFirstOccurrence (char unq [],
int len, char cx, char cy, char cz) {
      int i;
2:    for (i=0;i<len;i++) {
3:        if (unq[i]==cx){
4:            unq[i] = cz;
5:            // a bug, should be a break;
6:        }
7:        if (unq[i]==cy)){
8:            unq[i] = cz;
9:            // a bug, should be a break;
10:       }
11:   }
12: }
```

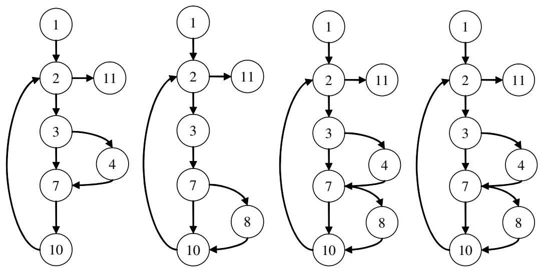


Fig. 1. Software Behavior Graphs of Four Executions.

The first two test cases result in correct execution traces, while the third and fourth result in failures. Fig. 1 shows the software behavior graphs of the four executions from the test cases. Each number in the graphs corresponds to the line number of the code. A discriminative graph is shown in Fig. 2. It highlights that the problem occurs when the basic block starting at line 4 is executed with that of line 8.

C. Two Limitations

Although the subgraph mining approach for bug signature identification has been shown effective through experimental evaluation on Siemens datasets in [2], we have observed that there are two limitations in the subgraph mining method, which affect the quality of the returned bug signatures.

The first limitation is that, the returned bug signatures are not minimal in size. A minimal bug signature is one which captures the bug and its context and nothing else. A signature which is not minimal will include unrelated program elements, thus making it harder for developers to localize the bug. Fig. 3(a) shows a bug signature which is discriminative but not minimal. This signature appears in the latter two graphs in Fig. 1 corresponding to failing runs but not in the correct runs, thus it is discriminative. But this is a larger signature than the smallest connected signature shown in Fig. 2.

The second limitation is that, the returned bug signatures

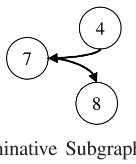


Fig. 2. A Discriminative Subgraph as Bug Signature.

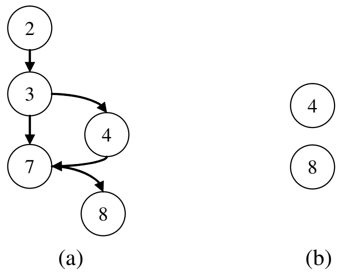


Fig. 3. (a) A Non-Minimal Discriminative Subgraph, and (b) A Minimal Disconnected Discriminative Subgraph.

can not capture a bug context which spans multiple program locations, as the algorithm mines connected subgraphs only. For the above program, the minimal signature is a disconnected graph as shown in Fig. 3(b). It states that nodes 4 and 8 co-occur together to discriminate failing from correct runs from the set of input traces.

In this work, we aim to address these two limitations. Accordingly, we have proposed two techniques of *signature minimization* (in Section IV) and *signature fusion* (in Section V) to achieve the above two goals. At first glance, it seems that minimization and fusion are achieving conflicting goals as one reduces the signature sizes while the other increases the sizes. Actually they are achieving two independent goals: the goal of signature minimization is to remove unrelated program elements from a bug signature and obtain minimal signatures with *the same* discriminative score; while the goal of signature fusion is to get signatures encompassing disconnected components so that a merged signature is *more* discriminative than each of the components itself.

IV. Signature Minimization

The best bug signature should be both relevant to the bug and minimal in size. A minimal bug signature captures the context that discriminates faulty from correct executions but excludes other program elements that appear in both faulty and correct executions. If the bug signature is too large, it impairs understanding and is less useful.

We propose two algorithms for bug signature minimization. Both algorithms take a bug signature and try to produce smaller signatures that have the same discriminative score as the larger one. The first is a complete algorithm which enumerates all subgraphs of the original signature and returns the smallest one with the same discriminative score. The second is a greedy algorithm that heuristically searches and returns a smaller bug signature that preserves the discriminative power if there is any.

Given a bug signature with n edges, the worst case complexity of the complete algorithm is $O(2^n)$, as it needs to enumerate all possible subgraphs of g. This is not scalable. Thus in the remainder of the paper we focus on the greedy algorithm. Details of the complete algorithm is available in the accompanying technical report [1].

The algorithm for greedy bug signature minimization is shown in Algorithm 1. The algorithm greedily removes one

Algorithm 1 MinimizeGraph: Greedy

```
Input: A signature graph $g$, discriminative score $F(g)$,
       correct and failing traces $D = \{(G_i, y_i)\}_{i=1}^{n}$
Output: One subgraph $g' \subset g$ with the same score

1: Let $E(g)$ be all edges in $g$;
2: Let $g_{min} = g$;
3: **for** $e \in E(g)$
4:     $g' = g_{min} \diamond e$;
5:     Calculate the discriminative score $F(g')$;
6:     **if** $F(g_{min}) == F(g')$
7:         $g_{min} = g'$;
8: **end for**
9: **return** $g_{min}$;
```

edge at a time (line 4) and evaluates the discriminative score (line 5). If the score of the smaller graph is the same, the current minimal signature g_{min} will be updated (lines 6-7) and the algorithm continues to remove edges of g_{min} in the *for* loop. The algorithm terminates when there is no further edge removal that is possible. Finally g_{min} is returned.

Note that g_{min} may not be the true minimal signature as Algorithm 1 does not consider all combinations of edge removals. Also, there might be multiple real minimal signatures; our heuristics returns only *one* signature that is of the same size or smaller with the same discriminative score. At line 3, the edges in E(g) are traversed in a predefined order. Due to the heuristics, if a different ordering is employed, it might produce a different minimal graph. The worst case complexity of Algorithm 1 is $O(n)$.

V. Signature Fusion

Signatures produced by [2] do not capture bug contexts encompassing multiple disconnected localities. However, a bug context may be separated by a common piece of code that is executed by both correct and faulty executions. In this situation, a graph mining algorithm would produce two discriminative subgraphs corresponding to the part of the program *before* and *after* the common code segment. However, it should be more informative for a debugger to understand and fix the bug, if the two discriminative subgraphs are merged into one bug signature that captures the overall context. To accomplish this goal, a bug signature fusion strategy can be designed to obtain discriminative bug signatures that span multiple disconnected locations.

The basic operation in signature fusion is to merge a pair of graphs to form a larger graph. The algorithm, called $FuseTwoGraphs$ is shown in Algorithm 2. The algorithm first merges the two input graph signatures to a larger one g_{fuse} (line 1) where the merge operation is denoted by '$\oplus$'. In a software behavior graph, each node has a unique label representing a unique function or basic block. This simplifies the merge as the resultant node set would simply be the union of the two node sets from the two input graphs; the resultant edge set would be the union of edges from the

Algorithm 2 FuseTwoGraphs

```
Input: Two signature graphs $g$ and $g'$,
       discriminative scores $F(g)$ and $F(g')$,
       correct and failing traces $D = \{(G_i, y_i)\}_{i=1}^{n}$
Output: Fused signature $g_{fuse}$ with a higher score or $null$

1: Let $g_{fuse} = g \oplus g'$;
2: Calculate the discriminative score $F(g_{fuse})$;
3: if $F(g_{fuse}) > \max(F(g), F(g'))$
4:     return $g_{fuse}$;
5: else
6:     return $null$;
```

two graphs. It then computes the discriminative score of the merged graph g_{fuse}. To compute this score, the algorithm needs to scan the graph database D and checks for the number of graphs to which g_{fuse} is subgraph isomorphic, i.e., $|\{G_i | G_i \in D \; and \; g_{fuse} \subset G_i\}|$. The discriminative score of g_{fuse} is then compared to that of its two constituent graphs. If the merged graph g_{fuse} has a higher discriminative score than the two graphs, then we return the fused signature (lines 3-4), otherwise no merge operation is performed and a $null$ value is returned (lines 5-6).

The function $FuseTwoGraphs$ is used as a building block to generate bug signatures with disconnected components. For the problem of signature fusion, again we have two alternative solutions: complete and greedy. The complete version enumerates all kinds of disconnected graphs by merging all possible combinations of connected graphs. It guarantees that the most discriminative bug signatures with disconnected components are found. The greedy version on the other hand is a best-effort solution. Several heuristics are used to greedily explore the promising search space that is likely to produce bug signatures with high discriminative scores.

In the complete algorithm, we need to first generate all possible distinct connected subgraphs that appear in the graph database as building blocks of the larger disconnected bug signatures. The number of such graphs is combinatorial to the size of the largest graph in the graph database. Each pair of connected graphs could then be fused using the $FuseTwoGraphs$ procedure. Such fused graphs can be further fused with other connected graphs in a recursive fashion. Thus this process is highly expensive, as in the worst case $2^{|CG|}$ fusion operations need to be performed, where $|CG|$ is the number of distinct connected subgraphs in the graph database. Hence, the approach can only work on a database consisting of a small number of graphs of small sizes, but can hardly scale to large graphs. In this work, we resort to the greedy algorithm for signature fusion. We employ the following heuristics to reduce the fusion cost:

1) Only merge graphs within the top-k discriminative graph set G_k. It is likely that a connected component of a highly discriminative bug signature is very discriminative itself.
2) Perform signature minimization before fusion. As a connected graph g may be fused with many different graphs to form multiple disconnected signatures, if we perform minimization first, we only need to examine g once for minimization. On the other hand, if we perform fusion first, the same minimization operation on g is then repeated many times on those fused signatures containing g.

In addition, we apply Lemma 1 to avoid many useless fusions.

Lemma 1: Consider G_k as the top-k most discriminative connected subgraphs with unique node labels. There is no need to merge two graphs $g_i, g_j \in G_k$ if $V(g_i) \bigcap V(g_j) \neq \emptyset$.

Proof: If any two graphs $g_i, g_j \in G_k$ share some common nodes, i.e., $V(g_i) \bigcap V(g_j) \neq \emptyset$, then the fused graph $g_{fuse} = g_i \oplus g_j$ is also connected through the common nodes. Since G_k contains the k most discriminative connected subgraphs and g_{fuse} is a connected subgraph, then either of the following two conclusions holds:

1) $g_{fuse} \in G_k$, if $F(g_{fuse}) \geq \min_{g \in G_k} F(g)$; or
2) $g_{fuse} \notin G_k$, if $F(g_{fuse}) < \min_{g \in G_k} F(g)$.

For either case, there is no need to merge the two graphs g_i and g_j as either it is already reported or would not be among the top-k most discriminative graphs. ■

Furthermore, we notice that more than two graphs could be fused together to form a larger signature. A complete exploration of all possible combinations of fusions even among the top-k graphs G_k would be prohibitively expensive. Thus we employ a greedy approach that greedily fuses graphs with the following heuristics:

1) If two graphs g_i and g_j could not be fused (i.e., the fusion results in a graph with a lower discriminative score), each of g_i and g_j is likely not fuse-able with a larger graph containing the other graph.
2) If two graphs g_i and g_j could be fused (i.e., the fusion results in a graph with a higher discriminative score), each of g_i and g_j is likely fuse-able with a larger graph containing the other graph.

Based on the above heuristics, we propose Algorithm 3 for the generation of bug signatures with disconnected components. The algorithm takes as input a list of minimized top-k connected signatures. It then iteratively tries to fuse one signature with another (lines 1-5). Based on Lemma 1, if two signatures share nodes, there is no need to fuse them (lines 3-4). If two signatures g_i, g_j are fused into a larger one, we add it to the end of G_k (lines 6-7), so that this fused graph $g_i \oplus g_j$ will be further fused with other graphs in G_k. In addition, we remove g_i and g_j from further consideration (line 8), as there exists the fused signature $g_i \oplus g_j$ containing them but with a higher discriminative score. According to the above heuristics, other graphs in G_k will be fused with the larger graph $g_i \oplus g_j$, but not with g_i or g_j. Finally we return a sorted list of signatures according to their discriminative scores in G_k (lines 9-10).

Note that g_{fuse} may not be the best set of fused signatures as Algorithm 3 does not consider all combinations of graph

Algorithm 3 Gen-Disconnected-Signature

```
Input: Top-$k$ connected signatures $G_k$,
       correct and failing traces $D = \{(G_i, y_i)\}_{i=1}^n$
Output: The set of fused signatures
1: for graph $g_i \in G_k$
2:     for graph $g_j \in G_k$ after $g_i$
3:         if $V(g_i) \bigcap V(g_j) \neq \emptyset$
4:             continue;
5:         $g_{fuse} = FuseTwoGraphs(g_i, g_j, F(g_i), F(g_j), D)$;
6:         if $g_{fuse} \neq null$
7:             Add $g_{fuse}$ to the tail of $G_k$;
8:             Remove $g_i, g_j$ from $G_k$;
9: Sort all $g \in G_k$ according to $F(g)$;
10:return $G_k$;
```

fusions. Also, there might be multiple best sets of fused signatures; our heuristics returns only *one* set of signatures, in which each fused signature has a higher discriminative score than the original signatures being fused. At line 1, the graphs in G_k are traversed in a predefined order. Due to the heuristics, if a different ordering is employed, it might produce a different set of fused signatures.

VI. An Integrated Solution

Based on the proposed signature minimization and fusion techniques, we design an integrated solution, called *MIN-FUSE* with the following steps.

1) Get the top-k bug signatures G_k returned by Top-K LEAP, where $k = 10$ in our experiments;
2) Get the top-10 most discriminative single events Evt_k that appear in the buggy executions;
3) Remove from G_k those signatures whose events appear in Evt_k with the same or higher discriminative score;
4) Let $C_0 = G_k \cup Evt_k$. Sort signatures in C_0 in the decreasing order of the discriminative score. If several signatures have the subgraph-supergraph relationship but have the same score, only keep the smallest one. The set after this processing is denoted as C;
5) Perform signature minimization on C;
6) Perform signature fusion on the minimized signatures. Two graphs are fused only if the resultant graph has a discriminative score at least 10% higher than the max of the two;
7) Sort the resultant signatures and return the top-k to users. In our experiments, we return top-10 signatures.

In step 2, we extract Evt_k, which are discriminative events from buggy executions, to enrich the bug signature set. The purpose is to compensate a restriction of Top-K LEAP, which does not mine or return single events as signatures. The smallest possible graph returned by Top-K LEAP has one edge and two endpoints. In step 3, if a signature has the same or lower score with one or more of its constituent events, the signature is removed. The other steps are quite self-explanatory.

VII. Experiments

In this section, we present our experiments to test the performance of the proposed signature minimization and fusion techniques. We experiment with the Siemens benchmark datasets which were developed by researchers of Siemens Corporation to test the adequacy of test coverage strategies [7]. These datasets are based on seven programs which are seeded by commonly found bugs. Each version of a program is seeded with one unique bug. These datasets have been used by various research studies on bug localization, e.g., [3], [9], [10] as well as bug signature identification [6], [2].

To complement the Siemens dataset that focuses on small programs and seeded bugs, we also investigated the Space dataset from Software-artifact Infrastructure Repository (SIR) [4]. Space is an interpreter of an array definition language (ADL) consisting of 9564 lines of code written in C. It checks a file that contains ADL statements for compliance and eventually outputs either an "array data file containing a list of array elements, positions, and excitations" or error messages [4]. Space comes with 38 versions, each of which has a single *real* fault discovered during the development of Space. It also comes with 13,585 test cases. As three versions (v1, v2, and v32) do not result in any faulty executions after running the test cases, we omit them from evaluation. Hence, in total we consider 35 versions of the Space program.

We compare our technique with Top-K LEAP, the state-of-the-art bug signature identification tool by Cheng et al. [2]. To measure the effectiveness of the mined bug signatures, we use the following two measures: the number of *bugs missed* and the number of *nodes investigated* to a relevant signature (inclusive of the relevant signature). The third author manually browsed through the returned bug signatures and marked whether each of the signatures (or contexts) is relevant or not. The measures of *the number of bugs missed* and *the number of nodes investigated* could then be computed.

A. Experimental Results

We reported the experiment results with the Siemens and Space datasets in the following paragraphs.

Siemens Dataset. In the following we show the results on the seven Siemens datasets. For each dataset we take the average of *the number of bugs missed* and *the number of nodes investigated* across all versions. We consider two levels of granularity of the program, based on tracing at the method and basic block levels respectively. The results for the method level for both Top-K LEAP and our method MIN-FUSE are shown in Table II. The corresponding results for the basic block level are shown in Table III. The columns *Bugs Missed* and *|Nodes| Examined* correspond to the number of bugs missed and the number of nodes traversed to find a relevant signature respectively.

For the method level, Top-K LEAP misses 26 bugs in total while MIN-FUSE misses only 11 bugs. Thus MIN-FUSE misses 57.7%fewer bugs on the Siemens datasets. The number of nodes traversed is on average 4.83 for Top-K LEAP and 3.95 for MIN-FUSE, demonstrating that MIN-FUSE achieves

TABLE II
RESULT – METHOD LEVEL (SIEMENS)

	Top-K LEAP		**MIN-FUSE**	
Program	Bugs Missed	|Nodes| Examined	Bugs Missed	|Nodes| Examined
tcas	2	4.66	6	3.77
print_tokens	0	3.57	0	2.57
print_tokens2	4	4.30	3	3.60
schedule	3	3.60	0	3.22
schedule2	7	0.60	2	5.00
tot_info	5	2.96	0	2.70
replace	5	8.50	0	5.38
Total Bugs/ Avg. Nodes	**26**	**4.83**	**11**	**3.95**

TABLE III
RESULT – BASIC BLOCK LEVEL (SIEMENS)

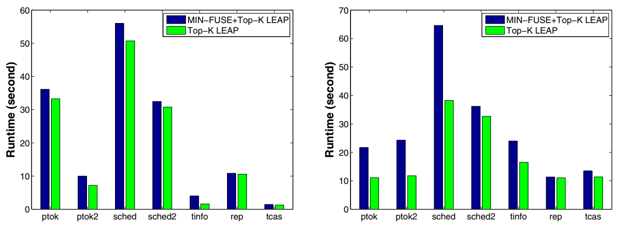
	Top-K LEAP		**MIN-FUSE**	
Program	Bugs Missed	|Nodes| Examined	Bugs Missed	|Nodes| Examined
tcas	0	5.71	0	4.12
print_tokens	0	4.71	0	1.57
print_tokens2	0	5.30	0	2.50
schedule	1	3.00	0	2.33
schedule2	4	8.80	3	5.70
tot_info	2	8.70	1	5.13
replace	4	12.47	3	5.72
Total Bugs/ Avg. Nodes	**11**	**7.83**	**7**	**4.42**

a reduction by 18.2%. Furthermore, we find at the method level, the program *schedule2* is the most difficult one for Top-K LEAP. However, using MIN-FUSE we could reduce the number of bugs missed from 7 to 2. There is an increase though in the number of nodes traversed for MIN-FUSE on *schedule2*. This is because in many versions of this program Top-K LEAP does not return any signature, hence contributing a signature of size 0.

For the basic block level, Top-K LEAP misses 11 bugs in total while MIN-FUSE misses only 7 bugs. Thus MIN-FUSE misses 36.4% fewer bugs. The number of nodes traversed is reduced by 43.6% with MIN-FUSE, as the number of basic blocks investigated is on average 7.83 for Top-K LEAP and 4.42 for MIN-FUSE. In addition, we find at the basic block level, the program *tot_info* is among the hardest ones for Top-K LEAP. However, using MIN-FUSE we could reduce the number of bugs missed from 2 to 1, along with a reduction in the number of nodes traversed by 41.0%.

Space Dataset. For space dataset, at the method level, the number of bugs missed is reduced from 27 to 25 (by 7.4%). The average number of nodes investigated to the relevant signature is reduced from 34.29 to 11.74 (by 65.8%). At the basic block level, the number of bugs missed is reduced from 9 to 8 (by 11.1%) and the average number of nodes investigated to the relevant signature is also steeply reduced from 34.77 to 5.00 (by 85.6%). This shows that our approach works well on larger programs with real bugs.

Runtime. From the experiments, we find that the cost of performing minimization and fusion is much smaller than



(a) Method Level (b) Basic Block Level

Fig. 4. Runtime Comparison

the cost of mining the top-k signatures in most cases or comparable in a few other cases. Fig. 4 shows the runtime comparison between our approach (i.e., MIN-FUSE+Top-K LEAP, as we ran MIN-FUSE on the Top-K LEAP results) and Top-K LEAP alone at the method and basic block levels of the Siemens dataset respectively. The average runtime at the method level of Space program is 498.23 seconds by MIN-FUSE+Top-K LEAP versus 472.77 seconds by Top-K LEAP (5.4% overhead). The average runtime at the basic block level of Space program is 5587.33 seconds by MIN-FUSE+Top-K LEAP versus 5421.29 seconds by Top-K LEAP (3.1% overhead). These results mean that we could effectively handle a potentially explosive search space on mining signatures with disconnected components with a reasonable additional cost. This shows the effectiveness of our search space pruning strategies, various heuristics, and the greedy algorithms.

Threats to Validity. Similar to other empirical studies, there are several threats to validity in interpreting the results. First, we ask the third author to label the bug signatures as being relevant or not. This process might be error prone as there are many signatures to be labeled. The signatures have been double checked and some have been further investigated to reduce errors. Second, we experimented with only seven small C programs in the Siemens test suite and a larger C program Space. Although our experiments utilized experiment benchmarks reported in many of the past studies on bug localization and bug signature identification on the types and scale of the programs investigated, we have not investigated the utility of the method on other types of programs aside from the eight programs. We have not experimented with non-C programs either. However since the signatures we mine make use of discriminative control flow information and the concept of control flow is generic, the result is likely to apply to other types of programs.

VIII. EXPERIENCE

In this section, we describe a bug signature demonstrating the benefit of signature minimization and fusion. Due to space limitation, we put other experiences in an accompanying technical report [1].

Program *replace* takes in a string s, a pattern p, and another string r. It should replace the occurrences of p in string s with r. In version 20 of program *replace* the bug is located in the function `esc` which is meant to handle the escape character (see Fig. 5). The statement marked as "`// fault`"

```
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:        m = amatch(lin, i, pat, 0);
2:        if ((m >= 0) && (lastm != m)) {
3:            putsub(lin, i, m, sub);
4:            lastm = m;
5:        }
6:        if ((m == -1) || (m == i))
7:            i = i + 1;
8:        else
9:            i = m;
}
```

Fig. 5. Code snippet from version 20 of *replace*.

should have been: "`result = ESCAPE`". From the function, we can see that the bug (line 6) is manifested when `ESCAPE` is followed by `ENDSTR` in the pattern p. Furthermore, the bug is only manifested in a particular context, namely when there is at least one occurrence of pattern p in string s. This check is made in function `subline`; the variable m would not be equal to -1 or i, if a match is found (line 9). Whenever this happens, a fault occurs. We return a bug signature comprising of two disconnected nodes corresponding to line 6 of procedure `esc` and line 9 of the procedure `subline`. The occurrence of each of the basic blocks alone does not reveal the fault. However, when the two occur together, the bug is manifested. This is an example of a context-sensitive or path-dependent bug. Bug signatures could provide information on the relationship between fault candidates. Such information, however, could not be provided by regular fault localization tools that return single statements.

For this case, Top-K LEAP splits the bug signature into two. Hence, it is harder to understand the overall context of the bug. As reported in [6], RAPID is able to capture a similar variant of the bug signature with a larger size: a 6-element signature including line 6 of `esc` and line 3 of `subline`. Thus, the signature reported by MIN-FUSE is smaller than that reported by RAPID, and both smaller and more informative than that by Top-K LEAP.

In effect our proposed method MIN-FUSE can capture all bug signatures reported in the experience sections of [6], [2]. In [2], Top-K LEAP reports two signatures which RAPID fails to identify. In [6], RAPID identifies a signature in full while Top-K LEAP splits it into several and reports them as different signatures.

IX. Conclusions and Future Work

In this paper, we extend the work on bug signature identification by employing a unique strategy of signature minimization and fusion. The state-of-the-art work on identifying bug signatures called Top-K LEAP extracts the top-k most discriminative connected signatures from two sets of graphs corresponding to buggy and correct executions. However, the returned graphs might not be minimal and could not capture signatures consisting of several disconnected parts. To address these limitations, first we employ a graph minimization strategy to capture smaller signatures. The large search space involved in minimizing graphs is avoided by employing a greedy heuristic solution. Next, we employ a signature fusion strategy to capture discriminative signatures with disconnected parts. We have experimented our solution on Siemens benchmark and a larger program called Space containing many real bugs. Our approach, called MIN-FUSE, outperforms Top-K LEAP in terms of the number of bugs missed and the average number of nodes traversed to the relevant signature. The results show that the number of bugs missed could be reduced by up to 57.7% (method level of Siemens), and the average number of nodes traversed could be reduced by up to 85.6% (basic block level of Space).

In the future, we plan to further improve the heuristics, include other information in addition to the control flow information, and further increase the number of case studies.

Acknowledgement. The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No.: CUHK 411310).

References

[1] "Bug signature minimization and fusion (technical report version)," www.mysmu.edu/faculty/davidlo/minfuse/mf-tr.pdf.

[2] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *ISSTA*, 2009.

[3] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE*, 2005.

[4] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, 2005.

[5] J. Han and M. Kamber, *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.

[6] H. Hsu, J. A. Jones, and A. Orso, "RAPID: Identifying bug signatures to support debugging activities," in *ASE*, 2008.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *ICSE*, 1994.

[8] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003.

[10] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: statistical model-based bug localization," in *ESEC/FSE*, 2005.

[11] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.

[12] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003.

[13] G. Tassey, "The economic impacts of inadequate infrastructure for software testing." *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.

[14] J. Wang and J. Han, "BIDE: Efficient mining of frequent closed sequences." in *ICDE*, 2004.

[15] S. Wang, D. Lo, L. Jiang, and Lucia, "Search-based fault localization," in *ASE*, 2011.

[16] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transaction on Software Engineering*, vol. 28, 2002.