Mining Succinct Predicated Bug Signatures

Chengnian Sun
School of Computing
National University of Singapore
suncn@comp.nus.edu.sg

Siau-Cheng Khoo
School of Computing
National University of Singapore
khoosc@comp.nus.edu.sg

ABSTRACT

A bug signature is a set of program elements highlighting the cause or effect of a bug, and provides contextual information for debugging. In order to mine a signature for a buggy program, two sets of execution profiles of the program, one capturing the correct execution and the other capturing the faulty, are examined to identify the program elements contrasting faulty from correct. Signatures solely consisting of control flow transitions have been investigated via discriminative sequence and graph mining algorithms. These signatures might be handicapped in cases where the effect of a bug is not manifested by any deviation in control flow transitions. In this paper, we introduce the notion of predicated bug signature that aims to enhance the predictive power of bug signatures by utilizing both data predicates and control-flow information. We introduce a novel “discriminative itemset generator” mining technique to generate succinct signatures which do not contain redundant or irrelevant program elements. Our case studies demonstrate that predicated signatures can hint at more scenarios of bugs where traditional control-flow signatures fail.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—debugging aids, diagnostics

General Terms

Experimentation, Reliability

Keywords

bug signature, statistical debugging, feature selection

1. INTRODUCTION

Debugging is a process to eliminate program defects. During a debugging session, a developer needs first to identify the location of the bug, then figure out its cause and finally fix it. As widely known, debugging is a painstaking activity in software development and maintenance phases, especially when the symptom (or the manifestation) of a bug is not right next to where the bug is triggered. For instance, a non-crashing defect produces a wrong output at the end of the execution, but the cause may be at the very beginning of the program. Such scenarios are likely to take developers much time to discover the cause from the symptom.

Bug signature identification is an automatic technique to infer the cause or effect of a bug. Different from fault localization, which outputs single suspicious program element each time for debugging, a bug signature can capture the bug context comprising multiple elements.

Two pioneering studies in bug signature discovery construct the profiles from running the buggy program against a test suite via different structures. Specifically, Hsu et al. [12] profile multiple traces of visited basic blocks from a buggy program, and perform sequence mining [26] to get common longest sequences in faulty executions as bug signatures. Hong et al. [5] curl the basic block sequences to form software behavior graphs and apply graph mining algorithm LEAP [27] to get discriminative subgraphs as bug signatures. The case studies in both papers have shown that bug signatures carry additional contextual information that further aids developer in comprehending the bugs.

On the other hand, these techniques operate on profiles that contain only control-flow information. In addition, the inferred bug signatures typically include redundant and/or irrelevant information. Reflecting on the outcome of these work, we hypothesize that:

1. The effectiveness of bug signatures in detecting bugs can be significantly enhanced if they are inferred from profiles containing predicated data information.

2. The quality of bug signatures can be significantly enhanced if it can be succinctly represented.

```
1     Ele* find_nth(List* f_list, int n) {
2         if (!f_list)
3             return NULL;
4         Ele* f_ele = f_list->first;
5     /*(-)BUG: f_list->first in the conditional
6     should be f_ele.*/
7         for (int i = 1; f_list->first && i < n; ++i)
8             f_ele = f_ele->next;
9         return f_ele;
10     }
```

Figure 1: Code Snippet of schedule with a Bug at Line 7

To illustrate this, we consider a concrete example presented in Figure 1. This piece of code is extracted from a buggy program schedule in Siemens benchmarks. Given a linked list f_list and an integer n, this function returns the n-th element in the list. The bug is that at line 7, f_list->first in for-loop test should be f_ele.
instead. Thus if the list is not empty and f_list->length < n − 1, f_ele becomes NULL after (n − 2) iterations; then in (n − 1)-th iteration, a dereferencing operation is performed on a NULL pointer, leading to a segmentation fault.

Figure 2 shows the control flow graph of this example, and the number prefixing each item in the graph identifies a unique basic block, a branch or a predicate. The bug can be tracked during program execution when f_ele becomes NULL after statement 10 (i.e. predicate 11), and i is still less than n after the subsequent assignment statement 15 (i.e. predicate 16). Thus {11, 16} is one good signature providing adequate contextual information for developers to fix this bug.

The conciseness of the predicated signature in this example becomes clear when we contrast it against the top ranked (control-flow) signature returned from LEAP, which is \{1, 6, 8, 13\}. While this signature includes the bug location f_list->first, it also contains irrelevant statements. First, statement 1 is the entry of this function, unconditionally appearing in every execution with an invocation to find_nth no matter whether the execution is correct or faulty. Second, statement 13 is an exit of this function, and it should appear in only correct executions as faulty executions should crash at statement 10 and cannot reach this statement.

In this paper, we propose a novel approach to automatically inferring bug signatures with two distinguished qualities: (1) they contain both data and control predicates; and (2) they are succinct in capturing bugs' cause and/or effect. Our case studies reveal that:

1. Data-predicated bug signatures possess high bug-predicative power. By comparing with control-flow-based signatures produced by LEAP, predicated bug signatures have much higher discriminative power (to be elaborated in the ensuing sections), and they can help in discovering a new class of bugs, the manifestation of which do not cause any control-flow deviation in the execution profiles.

2. Our novel bug signature mining algorithm, which is based on itemset generator mining approach, can perform much more efficiently than the state-of-the-art signature mining algorithm (aka., LEAP).

2. BACKGROUND

2.1 Overall Workflow

Figure 3 depicts the general workflow of bug signature mining. The initial inputs are a buggy program and a test suite. The buggy program is instrumented to collect runtime information. We run the instrumented version against the test suite to get a set of profiles. Based on the testing oracle, the collected profiles are classified into a correct and a faulty sets. Next the two sets are fed to our signature miner MPS to get a top-k ranked bug signatures based on their defect-predictive power.

![Figure 3: Overall Workflow to Bug Signature Identification](image)

We adopt the instrumentation scheme by Liblit et al. The following is a brief introduction. More information can be found in [17]

- **Branches.** At each conditional statement, each true or false branch is associated with a predicate to record whether this branch is taken at runtime.

- **Returns.** At each invocation site to a scalar-returning function, six predicates are created to capture relation between the returned value \(r\) and the constant \(0\): \(r \geq 0\), \(r > 0\), \(r = 0\), \(r \neq 0\), \(r < 0\) and \(r \leq 0\).

- **Scalar-pairs.** For each assignment to a scalar variable \(x = \cdots\), all same-typed in-scope variables and constant expressions are collected into a set \(V\). For each \(v \in V\), right after this assignment six predicates are created: \(x \geq v\), \(x > v\), \(x = v\), \(x \neq v\), \(x < v\) and \(x \leq v\).

- **Pointer-nullness.** For each assignment to a pointer variable \(p = \cdots\), a predicate \(p = \text{null}\) is created after this assignment.

The first instrumentation described above (predicates on conditionals) captures control-flow information, whereas the latter three record data-related information.

Note that our technique is general and the signature mining algorithm is orthogonal to the instrumentation scheme. More instrumentation types (aliasing among objects, def-use pairs, etc.) can be easily added without affecting the mining algorithm.

After running the instrumented program against the test suite, we get a list of profiles, each profile containing a set of predicates. We only retain those predicates which are captured to be true at runtime, and discard those that are unobserved or evaluated to false. Table 1 shows 5 profiles collected by running the buggy program in Figure 1. The column Input lists the test cases. Each test case is a pair, of which the first element is a list and the second is the parameter \(n\). The list could be null, empty (i.e. \([\]\)), or non-empty (e.g. \([1]\) containing one element, and \([1, 2, 3]\) containing three elements). The column Label marks the status of profiles, the plus (+ or positive) meaning that the profile corresponds to a correct execution, while the minus (− or negative) representing a faulty execution. The column Predicates shows all the predicates under observation in runtime. A bullet • in a cell \((i, j)\) means that the \(j\)-th predicate is evaluated to true in the execution corresponding to the \(i\)-th profile. Taking the profile \(t_2\) as an example, only two branches (4:false) and (12:false) are taken in runtime.

\(^1\)To simplify the illustration of our approach, we retain all branch predicates and remove all scalar-pair predicates except 11 and 16 in Figure 2. In the real case studies, our tool can precisely pinpoint \{11, 16\} among all predicates.
Table 1: Profiles Collected from Running the Buggy Program in Figure 1

<table>
<thead>
<tr>
<th>ID</th>
<th>Input</th>
<th>Label</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$(\text{null, 1})$</td>
<td>+</td>
<td>2:true 4:false 7:true 9:true 12:false 14:False 11: ( f_{\text{ele}} == \text{NULL} )? 16: ( i &lt; n )?</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$([1, 1])$</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$([1, 2])$</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$([1, 2, 3], 3)$</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$([1, 3], 1)$</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 2: An Example Database Constructed from Table 1

<table>
<thead>
<tr>
<th>ID</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>${2}, +$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>${4, 12}, +$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>${4, 7, 9, 14, 11}, +$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>${4, 7, 9, 14, 16}, +$</td>
</tr>
<tr>
<td>$t_5$</td>
<td>${4, 7, 9, 11, 16}, -$</td>
</tr>
</tbody>
</table>

2.2 Itemset Generator

We regard a profile as a set of items, each of which is a predicate observed to be true at runtime. With this itemset representation, we can therefore formulate signature identification as an itemset pattern mining task. Furthermore, as we aim to mine succinct signatures, we are particularly interested in itemset generator, a special pattern with minimal property. In this subsection, we provide an overview of the concepts and properties of itemset generators, using the profiles depicted in Table 1 as the running example.

Let \( I = \{i_1, i_2, \ldots, i_m\} \) be a set of distinct items, \( C = \{+,-\} \) be the set of the positive and negative class labels, and \( D \) be a database consisting of \( n \) transactions \( \{(T_1, c_1), \ldots, (T_n, c_n)\} \), where \( T_i \subseteq I \) and \( c_i \in C \). In the context of debugging, \( I \) corresponds to the set of predicates instrumented into the buggy program. The class label \( (+) \) identifies executions with correct output, whereas \( (-) \) identifies faulty executions. Each transaction is a profile consisting of predicates – a subset of \( I \).

Table 2 shows an example dataset constructed from the profiles of Table 1. \( I \) of this database is \( \{2, 4, 7, 9, 12, 14, 11, 16\} \). Every element identifies a predicate, for example, 2 stands for branch predicate (2:true), and 11 is data predicate (11: \( f_{\text{ele}} == \text{NULL} \)).

We define two classification functions \(+\) and \(-\) for a set of transactions \( S \),
\[
S^+ = \{(T, c) \in S | c = +\} \quad \text{and} \quad S^- = \{(T, c) \in S | c = -\}
\]
For example, \( D^+ \) or \( D^- \) denotes all the positive or negative transactions in \( D \) respectively. For an itemset (or pattern) \( P \subseteq I \), we define \( tx : 2^I \rightarrow 2^P \), returning all transactions in \( D \) containing pattern \( P \).
\[
\text{sup}(P) = |\{t \in D | (T, c) \in P\}|
\]
The support of \( P \) is defined as the number of transactions containing \( P \), i.e. \( \text{sup}(P) = |\{tx(P)\}| \) and \( \text{sup}^{-1}(P) = |\{tx(P)^+\}| \) and \( \text{sup}^-(P) = |\{tx(P)^-\}| \). The support of itemsets satisfies the following property stated in [2].

**Property 1 (APRIORI).** Given a pattern \( P \subseteq I \), \( \forall P' \subseteq I \), if \( P' \supset P \), then \( tx(P') \subseteq tx(P) \), and further
\[
\text{sup}^+(P') \leq \text{sup}^+(P) \text{ and sup}^-(P') \leq \text{sup}^-(P)
\]
Taking the database in Table 2 as an example, given a pattern \( \{4\} \) and its superset \( \{4, 7\} \), \( tx(\{4\}) \) returns transactions \( \{t_2, t_3, t_4, t_5\} \) and \( tx(\{4, 7\}) \) returns transactions \( \{t_3, t_4, t_5\} \), hence \( \text{sup}(\{4\}) < \text{sup}(\{4, 7\}) \).

**Definition 1 (Equivalence Relation).** Given an itemset database \( D \) defined over a set of items \( I \), the function \( tx : 2^I \rightarrow 2^D \) induces an equivalence relation \( \sim \) on \( 2^I \) such that for all itemsets \( P_1, P_2 \subseteq 2^I \), \( P_1 \sim P_2 \) if and only if \( \text{sup}(P_1) = \text{sup}(P_2) \). Furthermore, the equivalence class \( [P] \) of a pattern \( P \) is defined as \( \{P' \subseteq I | tx(P') = tx(P)\} \).

Thus, all patterns in an equivalence class are contained in the same set of transactions. In an equivalence class, all the minimal patterns are referred to as generators. Generators have the following property which has been proved in [15].

**Property 2 (GENERATOR).** A pattern \( P \) is a generator if and only if for every proper subset \( P' \subseteq P \), \( \text{sup}(P) < \text{sup}(P') \).

Figure 4 displays three equivalence classes in the database of Table 2. Each equivalence class is a lattice structure consisting of all patterns in the class. A node in a lattice represents a unique pattern. The links between nodes represent set relations superset and subset. The bottom nodes of a lattice are generators. The footnote below each lattice lists the support and transaction information. Figure 4a is the equivalence class of transactions \( \{t_4, t_5\} \), and \( \{16\} \) is the generator as its subset \( \emptyset \) has support \( \text{sup}^+(\emptyset) < 5 \) greater than \( \text{sup}^+(\{16\}) \). The same reason applies to Figure 4b and Figure 4c.

2.3 LEAP Signature Miner

In this paper, we compare our technique with the state-of-the-art signature miner LEAP proposed in [5], Hong et al. first profile each
program execution as a trace of basic blocks, then curl the basic block sequences to form software behavior graphs and apply graph mining algorithm LEAP [27] to get discriminative subgraphs as bug signatures, subgraphs which are frequently observed in faulty executions but rarely in correct ones. The discriminativeness of signatures is measured by information gain, which will be detailed in Equation 1.

\[ IG(p, n) = H(|\mathcal{D}^+|, |\mathcal{D}^-|) - \frac{p + n}{|\mathcal{D}|} \times H(p, n) - \frac{|\mathcal{D}| - (p + n)}{|\mathcal{D}|} \times H(|\mathcal{D}^+| - p, |\mathcal{D}^-| - n) \]  

where

\[ H(a, b) = - \frac{a}{a+b} \times \log_2\left( \frac{a}{a+b} \right) - \frac{b}{a+b} \times \log_2\left( \frac{b}{a+b} \right) \]

In the context of bug detection, however, we are only interested in patterns which are highly correlated with negative transactions (i.e. faulty executions), whereas the definition of \( IG \) is symmetric to some extent as a pattern highly correlated with positive transactions also carries high information gain. Henceforth, we leverage the notion of information gain to define the following discriminative significance measure:

\[ DS(p, n) = \begin{cases} IG(p, n) & \text{if } \frac{n}{|\mathcal{D}^-|} > \frac{p}{|\mathcal{D}^+|} \\ 0 & \text{otherwise} \end{cases} \]

Based on Definition 2, all the itemset generators in an equivalence class constitute a bug signature. As they have the same positive and negative supports, their \( DS \) values are also the same. Thus we use this \( DS \) value as the discriminative significance of a signature.

### 3.3 Top-k Bug Signatures

It is a good practice to examine bug signatures in the descending order of discriminative significance. Therefore we define the bug signature identification as a top-k discriminative pattern mining problem.

**Definition 3 (Mining Top-k Bug Signatures).** Given a labeled database \( \mathcal{D} \) constructed from faulty and correct profiles, and an integer \( k \), identify \( k \) signatures \( \{s_i\}_{i=1}^k \) from \( \mathcal{D} \), such that maximize \( \sum_{i=1}^{k} DS(tx(s_i)^+, |tx(s_i)^-|) \), where \( tx(s) = tx(g) \) for any \( g \in s \).

The notations \( tx(s_i)^+ \) and \( tx(s_i)^- \) denote the positive and negative transactions containing all generators in the signature \( s_i \), respectively. The following table shows top-5 signatures for the profiles in Table 1. The second column lists the detail supports. The last column shows the discriminative significance scores. As discussed earlier, the signature \( \{((111_f\_ele==NULL), (161\_c=n))\} \) is the best, for it has the highest discriminative significance score.

### 4. ALGORITHMS

We first review the data structure to mine frequent itemset generators, and then present our algorithm for discovering bug signatures via a novel discriminative generator mining algorithm.
4.1 Gr-tree to Mine Itemset Generators

Li et al. proposed a tree-based representation of transactions to efficiently mine frequent itemset generators [15]. We briefly describe this data structure in the section. Given a database \( db \) consisting of positive and negative transactions, a Gr-tree is a compact representation of \( db \), denoted as a tuple \( GTree_{db}^{pos/neg} \), where \( prefix \) is an itemset prefixing all the items in the Gr-tree. For the original database \( D \), \( prefix \) is \( \emptyset \). Figure 5 shows the Gr-tree of database \( db \) in Table 2, \( GTree_{db}^{0} \). Each Gr-tree has a head table, storing all items in descending order of their supports. If two items have the same support, then they are sorted randomly, e.g., items 11, 14 and 16.

A Gr-tree has the following two properties:

1. It does not store items of very low negative support. This holds because such items will not contribute to good and discriminative bug signatures.

2. It does not store items which have full support (i.e., which occur in all transactions in the database \( db \)). This holds since such items cannot be a part of any generator, as proven in [15].

Specifically, the second property enables compact representation of database and efficient discovery of generators.

Each item in the table has a link to its corresponding nodes in the tree. A path from root to a node \( m.t.(+p,-n) \) represents an itemset pattern comprising of the items in the path, supported by \( p \) positive and \( n \) negative transactions. For example, the path \( (root, 4.(+3,-1), 7.(+2,-1), 9.(+2,-1), 11.(+1,-1), 16.(+0,-1)) \) represents a pattern \( \{4, 7, 9, 11, 16\} \) appearing in no positive and 1 negative transaction.

Generators are developed by recursively adding a new item into the developing generator, and creating a conditional database of transactions wrt. this new item.

**Definition 4 (Conditional Database).** Given a Gr-tree \( GTree_{db}^{0} \), let \( a_1, \ldots, a_n \) be items in its head table. Then the conditional database of \( a_i (1 \leq i \leq n) \) is denoted by \( CD_{GTree_{db}^{0}}^{pos/(a_i)} \), as the set of path segments exclusively between the root and \( a_i \) for all paths containing \( a_i \).

A conditional database \( CD_{GTree_{db}^{0}}^{pos/(a_i)} \) is a projection of the original database obtained by only selecting transactions containing the pattern \( px \cup \{a_i\} \), and removing \( a_i \) and items below \( a_i \) in the head table of the Gr-tree.

The table above shows the conditional database of \( GTree_{db}^{0} \) in Figure 5 with respect to item 16, \( CD_{GTree_{db}^{0}}^{pos/(16)} \). In order to construct this conditional database from \( GTree_{db}^{0} \), we first extract all path segments between root and nodes 16:

![Figure 5: Gr-tree of the Database of Table 2 with prefix = \( \emptyset \)](image)

Table 4: The Conditional Database of Figure 5 w.r.t Item 16

<table>
<thead>
<tr>
<th>ID</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_4 )</td>
<td>( {4,7,9,14}, + )</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>( {4,7,9,11}, - )</td>
</tr>
</tbody>
</table>

1. \( (root, 4.(+3,-1), 7.(+2,-1), 9.(+2,-1), 11.(+1,-1), 16.(+0,-1)) \)
2. \( (root, 4.(+3,-1), 7.(+2,-1), 9.(+2,-1), 14.(+1,-0), 16.(+1,-0)) \)

As mentioned before, each path segment is an itemset, of which the support is that of the last item, so the support of all items in the first path segment is \( (+0,-1) \), and the support in the second segment is \( (+1,-0) \). We then remove root and nodes 16 from the segments and form the conditional database \( CD_{GTree_{db}^{0}}^{pos/(16)} \). Then we can build its Gr-tree, \( GTree_{cd}^{(-1)(16)} \) where \( cd = CD_{GTree_{db}^{0}}^{pos/(16)} \) shown in the figure below.

![Figure 6: Gr-tree of the Conditional Database of Table 4 with prefix = \{16\}](image)

Note that items 4, 7, 9 are removed from Gr-tree, as they have full support in the conditional database \( CD_{GTree_{db}^{0}}^{pos/(16)} \).

4.2 Algorithm Skeleton

Algorithm 1 outlines our signature mining technique. \( GS \) contains a list of tuples, from which top-k signatures are selected. Each tuple is of the form \( ((p, n), gens) \), where \( gens = \{g_i\}^m \) is a set of generators, each having support \( (+p, -n) \). The top-k signatures are identified in two steps, as shown in statements 2 and 3 of the algorithm:

**Algorithm 1: MineSignatures\( (D, k, neg\_sup, size\_limit) \)**

**Input:** \( D \), an itemset database constructed from profiles
**Input:** \( k \), the number of top discriminative signatures to mine
**Input:** \( neg\_sup \), signatures should be in at least \( neg\_sup \) faulty profiles
**Output:** \( RS \), a list containing top-k discriminative signatures

1. \( GS := []; \)
2. \( MineRec(GTree_{db}^{0}, k, neg\_sup, size\_limit, GS); \)
3. \( RS := ClusterGeneratorsIntoEquivalenceClass(GS); \)

**Step 1.** We mine \( k \) sets of generators. Each set is associated with a distinct support \( (+p, -n) \), and the generators in each set are of the same support \( (+p, -n) \). Also, the \( k \) sets have the top-k discriminative significance based on their \( DS(p, n) \). This step is done by calling \( MineRec \) at line 2. \( MineRec \) takes a Gr-tree as the first input; for the original database \( D \) we create its Gr-tree with empty prefix, and pass \( GTree_{db}^{0} \) to \( MineRec \). The last argument, \( GS \), stores the mined generators returned from \( MineRec \).

**Step 2.** We construct the top-k signatures by clustering generators into their equivalence classes. Upon reaching line 3, \( GS \) stores a list of tuples \( ((p, n), gens) \), each of which is a set of generators \( gens \) sharing the same positive and negative supports \( p \) and \( n \). However having the same supports is only a necessary condition for generators to be in the same equivalence class. Based on the definition of
bug signatures in Definition 2, they are not bug signatures yet. So at line 3, for each generator gen in GS, we scan the profile database D to compute tx(gen), and cluster all generators into their corresponding equivalence classes based on tx(gen). We store these equivalence classes to RS in descending order of discriminative significance. As two generators occurring in a tuple in GS may correspond to multiple equivalence classes, RS may have more than k signatures, and we proceed to only keep the top-k ones. Since developers are usually interested in a small set of bug signatures, the overhead of the clustering is usually low.

The efficiency and effectiveness of Algorithm 1 can be directly controlled by two of its parameters: neg_sup and size_limit. The parameter neg_sup sets the negative support threshold, thus requiring mined signatures to be present in at least neg_sup faulty profiles. Computationally, this allows Apriori property (cf. Property 1) to be exploited to avoid unnecessary computation time spent on constructing itemset patterns with too few negative supports.

The parameter size_limit enables us to cap the size of generators in bug signatures. As will be described in the next subsection, this setting confines the maximum depth of search space exploration, controlling the mining overhead. This is also useful when we want the bug location to be “approximated”, in trading off for efficiency. Take for instance the signatures shown in Table 3, if we set size_limit to 1, we obtain (11:f ele == NULL) as top-1, pointing to the general cause of the crashing failure. However, if we relax size_limit, we get (11:f ele == NULL) and (16:i < n), which is more specific.

4.3 Mining Discriminative Generators

Algorithm 2, MineRec, is inspired by the frequent generator mining algorithm described in [15], with two major differences. First, whereas the original algorithm aims to mine all frequent generators above a given support threshold, MineRec focuses only on the top-k sets of generators. Second, MineRec takes into account the discriminative power of the generator currently under investigation, and aggressively prunes the search space in a branch and bound fashion.

MineRec takes as input a Gr-tree tree. Recall that tree is rooted with a prefix pattern. MineRec outputs generators, which are grown by combining items in tree with prefix. It does so by performing depth-first-search over the pattern space. Specially, as shown between lines 2 and 4, MineRec first combines the prefix with each item in the head table of tree. Between lines 7 and 14, MineRec constructs, from each of the extended patterns, a conditional database db′ from tree, and then builds a smaller Gr-tree tree′ for db′. Lastly, MineRec calls itself recursively to discover qualified itemset patterns involving the respective extended pattern.

The branch and bound technique is employed (in line 12) to avoid making futile recursive invocation. Here, MinDS(GS) denotes the minimum discriminative significance of the generators in GS. If it is found that the new Gr-tree tree′ cannot output any generators with higher DS value (tested by calling the function UpperBound(tree′)) than the minimum discriminative significance in GS, and the size of GS is already k, then MineRec stops searching along this branch, and starts working on the next available pattern.

There are three early exits of this algorithm. At line 1, the function returns if tree is empty, as the current search path ends at tree. At line 5, if the size of generators mined between lines 2 and 4 equals to size_limit, then the function can safely stop, as MineRec outputs generators from small to big, and any generators mined in the future along the current path must be of a bigger size than size_limit. The third exit is at line 6 if tree is only a single path, as all possible generators derivable from tree have been mined between lines 2 and 4.

4.3.1 Computing UpperBound(tree)

For a pattern P, it has been studied that the information gain of super patterns of P is bounded by a formula over the support of P [6, 20]. Assume that we know a set of transactions uz(P) ⊆ tx(P) which contains all super patterns of interest P ⊇ P, referred to as unavoidable transactions.

ux(P) = ∩ P⊇P tx(P)

The information gain for any super pattern of P is upper bounded by the following formula, as in [20].

max{IG(sup+(P),|ux(P)|), IG(|ux(P)|+, sup−(P))}

In the context of bug detection, we focus on patterns which appear more frequently in negative transactions than positive ones, and therefore introduce a new upper bound for DS, as shown in the following theorem:

**Theorem 1 (Upper Bound of DS).** Given a pattern P, the discriminative significance of all its qualified super patterns is upper bounded by the following formula:

UB(P) = \begin{cases} IG(|ux(P)|+, sup−(P)) & \text{if } \frac{\text{sup}−(P)}{\text{DS}−} > \frac{|ux(P)|}{|DS|} \\ 0 & \text{otherwise} \end{cases}

Given a Gr-tree tree, the term “qualified super patterns” refers to all patterns that are derivable from tree. Furthermore, the DS

\footnote{The proof is presented in Section 9.}
upper bound of all these patterns (i.e., $\text{UpperBound}(\text{tree})$ invoked at lines 12 in Algorithm 2) can be computed as follows. We iterate every path of $\text{tree}$ from root to leaf nodes, until we find a path $(\text{root}, a(\pm m, -n), \cdots, \text{ln}(\pm q, -p))$ containing all items in the head table. The transactions containing $\{a, \cdots, \text{ln}\}$ form the unavoidable transactions of the prefix $px$. Since we only need to know the numbers of positive and negative unavoidable transactions to compute $UB(px)$, we can simply get them from the attached support information $(+p, -q)$ of the leaf node $\text{ln}$, namely

$$|px(\text{px})^+| = p \quad \text{and} \quad |px(\text{px})^-| = q$$

At last, we compute $UB(px)$ as the $DS$ upper bound of $\text{tree}$.

5. CASE STUDIES

We have implemented the proposed technique in a prototype named MPS (Mining Predicated Bug Signatures) in C++, and have experimented it with 75 faults in 5 buggy programs (i.e., $\text{print\_tokens}$, Unix utilities and $\text{space}$ interpreter) on a PC with Intel Core 2 Quad CPU 3.0GHz and 8GB memory. The $\text{print\_tokens}$ is a subject in Siemens benchmark which was developed to test the testing coverage strategies [13]. The programs $\text{grep}$, $\text{gzip}$, $\text{sed}$ are Unix utility programs of moderate size, and the last subject $\text{space}$ is an interpreter for an array definition language. The sizes of the programs range from 726 to 14,427.

<table>
<thead>
<tr>
<th>Subject</th>
<th>LoC</th>
<th>#Test Cases</th>
<th>#Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{print_tokens}$</td>
<td>726</td>
<td>4130</td>
<td>7</td>
</tr>
<tr>
<td>$\text{grep}$</td>
<td>10,068</td>
<td>199</td>
<td>12</td>
</tr>
<tr>
<td>$\text{gzip}$</td>
<td>5,680</td>
<td>214</td>
<td>16</td>
</tr>
<tr>
<td>$\text{sed}$</td>
<td>14,427</td>
<td>360</td>
<td>9</td>
</tr>
<tr>
<td>$\text{space}$</td>
<td>6,199</td>
<td>13385</td>
<td>31</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td>75</td>
</tr>
</tbody>
</table>

Each subject program has multiple versions and each version has a different bug. Table 5 shows the detail of these programs, including names in column Subject, size in column LoC, number of test cases in column #Test Cases, and number of faults per subject in column #Faults. The CBI sampler\(^3\) is used to instrument programs to collect data predicates.

We compare MPS with LEAP [5] in two modes: inter-procedural signature mining and intra-procedural signature mining. In the first mode, a bug signature is identified over the whole program, and the items in each generator in a signature can span across multiple functions. In the latter mode, the signature mining is performed repeatedly for each function, and the top-$k$ signatures are retained among all the signatures. Different from the first mode, items in a generator in such a signature must reside within the same function. Thus, in our case studies, we have four signature mining algorithms, i.e., $\text{mps\_inter}$, $\text{mps\_intra}$, $\text{leap\_inter}$ and $\text{leap\_intra}$, and use them to identify the top-1 bug signature. For MPS, we set the parameters $\text{neg\_sup} = 0.5^4$ and $\text{size\_limit} = 2$. Our experiments show that MPS outperforms LEAP in both modes, in terms of its mining speed and the quality of signatures discovered.

Our case studies are designed as follows: 1) We conduct an objective comparison between discriminative significance values of the top-1 signatures produced by MPS and LEAP, thus measuring their ability in contrasting faulty executions from correct ones. 2) We use the measurement method similar to the computation of $\text{DS}$ described in [23, 9] to compute $\text{DS}$. Since we only need to know the numbers of positive and negative transactions to compute $\text{DS}$ of a signature, we can simply get them from the attached support information $(+p, -q)$ of the leaf node $\text{ln}$, namely

$$|px(px)^+| = p \quad \text{and} \quad |px(px)^-| = q$$

At last, we compute $\text{DS}(px)$ as the upper bound of $\text{tree}$.

### 5.1 Objective Comparison with LEAP

A signature with higher discriminative significance indicates a higher correlation with the faulty executions, namely, it appears in more faulty executions yet fewer correct executions. As such, the signature may carry more predictive power in highlighting the bug. Thus, in this experiment, we use information gain (IG) as an objective metric to evaluate the performance of MPS and LEAP. Specifically, we measure the absolute improvement of MPS over LEAP in terms of IG scores of top-1 signatures, which is defined as $(\text{IG}_{\text{MPS}} - \text{IG}_{\text{LEAP}})$.

IG is widely used in information theory and machine learning [22]. For example, it is used to measure the change in information entropy from a prior state to a state that takes some information; in general classification algorithm, it is used to measure the effectiveness of features in classifying un-labeled examples. More importantly, in fault localization research, Lucia et al. have shown that IG is one of the best metrics in localizing bugs [19]. Alternatively, we can also use discriminative significance DS, instead of IG, in our comparison. Regardless of which of these two is used, the experiment outcomes are similar.

Table 6 shows the improvement of MPS over LEAP in different mode combinations. The first column indicates the modes of MPS and LEAP; the second and the third columns list the mean and the median of the absolute improvement. We also performed Wilcoxon signed-rank test for each combination, which yielded ($p < 0.0001$) throughout. Thus the improvement is statistically significant.

The improvement is due to the following reasons. First, the pruning technique used in MPS is based on the sound upper bound of $\text{DS}$ (cf. Theorem 1), whereas the pruning heuristics of LEAP are unsafe. Second, the predicates used in MPS provide more information on program states for characterizing bugs, especially useful for those which do not lead to any control-flow deviation from correct executions.

### 5.2 Proximity to Actual Bug

In this section, we measures the distance between the actual bug in the program and the signature mined. It aims to determine how far should the programmer go beyond the signature to localize the bug. The measurement method is similar to the computation of score described in [23, 9]. Specifically, the distance measure is performed on the program dependence graph (PDG) of the faulty version of the program. Given the actual bug and a signature, we identify all the corresponding nodes in the PDG signifying the actual bug and the signature. Let’s denote the bug node by $b$ and the latter set of signature nodes by $S$ respectively.

Let $k(n, e)$ be the set of nodes that are reachable in PDG from $n$ within the distance $e$. Through this, we determine the minimum
Table 7: Proximity Results

<table>
<thead>
<tr>
<th>subject</th>
<th>LEAP</th>
<th></th>
<th>MPS</th>
<th></th>
<th>LEAP</th>
<th></th>
<th>MPS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inter</td>
<td>ntra</td>
<td>inter</td>
<td>impr.</td>
<td>mtra</td>
<td>impr.</td>
<td>inter</td>
<td>ntra</td>
</tr>
<tr>
<td>print_tokens</td>
<td>0.718</td>
<td>0.859</td>
<td>0.939</td>
<td>11.7%</td>
<td>0.959</td>
<td>11.7%</td>
<td>0.645</td>
<td>0.809</td>
</tr>
<tr>
<td>grep</td>
<td>0.505</td>
<td>0.358</td>
<td>0.912</td>
<td>80.5%</td>
<td>0.889</td>
<td>76.1%</td>
<td>0.590</td>
<td>0.335</td>
</tr>
<tr>
<td>gzip</td>
<td>0.003</td>
<td>0.688</td>
<td>0.898</td>
<td>30.5%</td>
<td>0.891</td>
<td>29.5%</td>
<td>0.224</td>
<td>0.919</td>
</tr>
<tr>
<td>sed</td>
<td>0.648</td>
<td>0.692</td>
<td>0.919</td>
<td>32.9%</td>
<td>0.952</td>
<td>37.6%</td>
<td>0.518</td>
<td>0.635</td>
</tr>
<tr>
<td>space</td>
<td>0.000</td>
<td>0.000</td>
<td>0.992</td>
<td>0.000</td>
<td>0.996</td>
<td>0.000</td>
<td>0.000</td>
<td>0.053</td>
</tr>
</tbody>
</table>

Thus the undirected path between the median of Columns 6 and 7 can be interpreted similarly only except they list and Column 5 is the relative improvement of mps-inter and columns 8–13 list the mean. These two column sets proximity results for each program subject. Columns 2–7 list the X and compute expressed as a fraction of the PDG:

\[
N = \bigcup_{n \in S} k(n, d(n))
\]

to represent the maximum number of nodes/locations in the program a programmer has to examine, starting from node \( n \) in the signature. If \( d(n) = 0 \), the actual bug falls in the signature, and the number of nodes a programmer needs to examine is simply the size of the signature itself.

The fewer nodes a programmer must examine when locating a bug, the better the quality of the signature. This proximity is expressed as a fraction of the PDG:

\[
\lambda = 1 - \frac{|N|}{|PDG|}
\]

We used CODESURFER to compute the PDG of the buggy program, and compute \( \lambda \) for each of the signatures. Table 7 shows the proximity results for each program subject. Columns 2–7 list the median and columns 8–13 list the mean. These two column sets have the same structure, so we elaborate how to interpret the median columns. Columns 2 and 3 are the medians of the top-1 signatures produced by leap-inter and leap-intra. Column 4 is the median for mps-inter and Column 5 is the relative improvement of mps-inter over the best proximity value of leap-inter and leap-intra, i.e.,

\[
\frac{\text{mps-inter} - \max(\text{leap-inter, leap-intra})}{\max(\text{leap-inter, leap-intra})}
\]

Columns 6 and 7 can be interpreted similarly only except they list the median of mps-intra and its relative improvement over LEAP.

Overall, the improvement ranges from 11.7% to 1559.6%. In particular, the medians of leap-inter and leap-intra for the space subject are both zero: Since the number of profiles of space is large (i.e., 13585), each profile is also a big graph and sub-graph isomorphism checking used in LEAP is NP-complete, thus making leap-inter not able to terminate. In the case of leap-intra, due to the unsafe pruning heuristics in LEAP, it usually produces no signatures for space. We performed Wilcoxon signed-rank one-tail test between MPS and LEAP, and validated that the improvement of MPS is statistically significant with \( p < 0.001 \).

Moreover, it is interesting to see that signatures produced by leap-intra and leap-inter are comparable in proximity, so are mps-intra and mps-inter. As demonstrated in the following section, intra-procedural mining is significantly faster than inter-procedural mining, hence in practice, mps-intra can be used first to get quick diagnostic information. If the information is not enough, mps-inter can be invoked to provide alternative signatures.

5.3 Efficiency

Table 8 displays the runtimes of the four miners. The fourth column impr. lists the relative performance improvement of mps-inter over leap-inter, and the last column impr. lists the improvement of mps-intra over leap-intra. It shows clearly that except for gzip in inter-procedural mode, MPS is much superior to both the LEAP variants in terms of speed; furthermore, mps-intra is significantly faster than the other three by 140.46% – 12800%.

5.4 A Debugging Session for sed

This section describes how we use MPS to debug a fault in the sed program. The bug is at line 4, where the operator \( <= \) should be \( < \) instead. This bug causes the program to read the terminating null-character ‘\0’ of the input string, an unexpected behavior. The
### 5.5 A Debugging Session for tcas

The following describes a debugging session we conducted with MPS on the Siemens benchmark. In version 3 of tcas program, the function `alt_sep_test` has a bug at line 7, where the operator `||` should be `&&`. MPS outputs a signature including two predicates for this bug:

\[
(ch < 1) \text{ at line 5, and } (\text{opt} == 'e') \text{ at line 16}
\]

These two precisely capture the context under which the bug manifests itself. In contrast, LEAP outputs a signature containing four branches in a function, of which three conditions involve the character returned by `inchar()`. However, all these branches are far from the bug location, and not related to the bug. Worse still, that `inchar()` is intensively used in that function, and it is difficult to figure out the difference between calls in the signature and the other calls. In terms of discriminative significance, the MPS signature appears in 0 correct and 18 faulty executions with `DS = 0.0403`, whereas ours appears in 1 correct and 19 faulty executions with `DS = 0.08`.

### 5.6 Threats to Validity

As an empirical study, our experimental results are subject to two threats to validity. First, threats to construct validity concern whether the metrics used in the evaluation of MPS are proper. In this paper, we use information gain and proximity to measure the performance of MPS. The first one has been shown to be a good metric for fault localization in [19], and the second one is also widely used in debugging research projects. Both metrics are objective. The first one characterizes the capability of signatures in contrasting faulty executions from correct ones. The latter one mimics the developers’ behavior in debugging, measuring not only the effort required to figure out the cause of the bug starting from the mined signatures.

With regard to the concern that our results might not generalize to broader population of programs, we note that our algorithm assumes that the manifestation of a bug is highly correlated with a set of predicates; the applicability of this assumption to buggy programs in general is commonly accepted by the research community. As far as scalability is concerned, the runtime performance of `mps-inter` may degrade with large sets of profiles. However `mps-intra` is not affected much and can output signatures of comparable quality.

### 6. RELATED WORK

This section surveys and classifies research studies related to our work in three categories: bug signature mining, fault localization, and discriminative pattern mining. **Bug Signature Mining.** As pointed out by Hsu et al [12] and Parnin et al. [21], in the absence of the context in which a bug occurs, it is difficult for developers to conduct a debugging session. Hsu et al. utilize BIDE [26] sequence miner to discover longest common subsequences as bug signatures from a sequence database consisting of suspicious program statements in [12]. Cheng et al. [5] curl sequences into software behavior graphs and employ LEAP [27] to observed from the second predicate of the signature). In order for the execution to reach the statement at line 14, the test at the if statement at lines 11 and 12 must be true. Since `tcas_equipped` is known to be true, we therefore infer that `intent_not_known` must be true. At this point, we can ask if it is reasonable to set `alt_sep` to 1 when the "intent is not known". If it is reasonable, we can continue our investigation to check when `intent_not_known` is set to true. Based on the assignment to `intent_not_known` at line 6, the fact that `intent_not_known` must be true, and the first predicate in the signature, we can infer that in faulty profiles `Two_of_Three_Reports_Valid` and `(Other_RAC == 0)` cannot be true at the same time, and then question if it is reasonable to set `intent_not_known` to `true` in this situation. We thus arrive at the source of the bug.

On the other hand, for this version, LEAP outputs a large subgraph containing 21 basic blocks in functions `main`, `alt_sep_test` and another two, many of which do not help, but act as deterrence to the debugging process, in our opinion. Lastly, in terms of discriminative significance, the signature obtained from LEAP is contained in 0 correct and 10 faulty profiles and its `DS` is 0.0403, whereas ours appears in 1 correct and 19 faulty profiles with `DS = 0.08`.

### Table 8: Runtime Statistics (in seconds)

<table>
<thead>
<tr>
<th>subject</th>
<th>mps-inter</th>
<th>leap-inter</th>
<th>impr.</th>
<th>mps-intra</th>
<th>leap-intra</th>
<th>impr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>50.32</td>
<td>76.38</td>
<td>51.79%</td>
<td>11.45</td>
<td>27.51</td>
<td>140.46%</td>
</tr>
<tr>
<td>grep</td>
<td>18.79</td>
<td>77.28</td>
<td>311.18%</td>
<td>0.30</td>
<td>38.75</td>
<td>12800%</td>
</tr>
<tr>
<td>gzip</td>
<td>131.44</td>
<td>31.71</td>
<td>-75.88%</td>
<td>1.35</td>
<td>28.29</td>
<td>1722.4%</td>
</tr>
<tr>
<td>sed</td>
<td>82.30</td>
<td>104.94</td>
<td>27.51%</td>
<td>5.66</td>
<td>51.31</td>
<td>806.30%</td>
</tr>
<tr>
<td>space</td>
<td>78.40</td>
<td></td>
<td></td>
<td>-</td>
<td>6.93</td>
<td>208.37</td>
</tr>
</tbody>
</table>

Figure 8: A Bug in Version 3 of tcas
discover discriminative subgraphs as bug signatures. The experimental results have shown that their approach outperforms RAPID [12]. Extending LEAP to discover *predicated* bug signatures can however be non-trivial. It is unclear how predicates should be encoded into the graph models: Encoding predicates as edges creates multi-edge graph, the mining of which will require non-trivial extension to LEAP; encoding predicates as nodes may create conflict with the pruning heuristics deployed by LEAP.

**Fault Localization.** In spectrum-based fault localization, program profiles or spectra obtained from faulty and correct executions are analyzed to locate bugs. Reniers and Reiss compare a failed execution with the nearest correct execution to locate suspicious program elements [23]. Liblit et al., Chao et al. and Zhang et al. find predicates that are correlated with failures [16, 28, 10]. Jones and Harrold use Tarantula [14], and Abreu utilize Ochiai [1] to rank suspicious program statements. Nainar et al. identify compound boolean predicates of size 2 for statistical debugging [3]. Chilimbi et al. use statistic metric to rank program paths which are correlated to bugs [8]. Differently, our approach targets at minimum bug signatures, which is capable of capturing multiple profile elements for bug diagnosis. It is also flexible: in case the location of a bug is highly discriminative, the location will be directly returned. In comparison with [3], we propose a systematic algorithm to mine succinct signatures of arbitrary size; Baah et al. propose a probabilistic program dependence graph to software fault localization, of which the probabilities are inferred based on observational studies and causal effect estimation [4]. Gore et al. recently study the reduction of confounding bias in predicate-level statistical debugging [11]. Our approach is orthogonal to these causal-inference-centric research [4, 11], and it will be interesting to integrate these techniques to tackle confounding bias at signature level. Rossler et al. combines statistical debugging and test case generation to isolate failure causes [24].

**Discriminative Pattern Mining.** Hong et al. mine discriminative pattern mining [53x352] itemset patterns based on information gain in [7]. Nijssen et al. transform discriminative itemset mining into a constraint satisfaction problem [20]. Lo et al. mine discriminative sequential patterns for software behavior classification [18]. Yan et al. mine discriminative subgraph patterns via leap search [27]. Sun et al. mine contrasting patterns for software process evaluation [25].

In comparison, our novel algorithm aims to mine discriminative itemset generators by contrasting faulty execution profiles from correct ones. Consequently, we improve on scalability in signature mining through avoiding construction of connected subgraph, and filter redundant information in signatures through mining of generators.

### 7. CONCLUSION AND FUTURE WORK

Understanding program bugs invariably involves reasoning through sequences of program states, which are typically represented by both data predicates and conditions (for directing control flow). Automatically identifying appropriate data predicates that represent either the cause or effect of a bug is a non-trivial task. Specifically, it can be challenging to extend the current control-flow based signature (generated by LEAP) to include such predicates.

In this paper, we propose a novel algorithm to automatically identify bug signatures consisting of data predicates and control-flow information. Compared to LEAP, our algorithm is sound, as the technique employed to prune the search space is safe, in information theoretic sense. With the presence of data predicates, the functional-