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*

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491449

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.

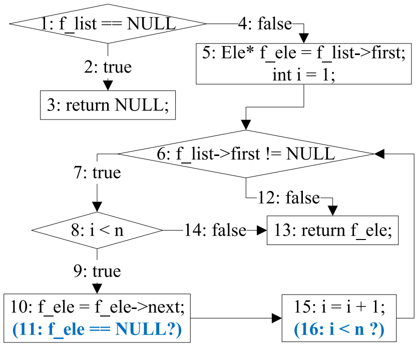

Figure 2: Control Flow Graph of Figure 1

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 efficient 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.

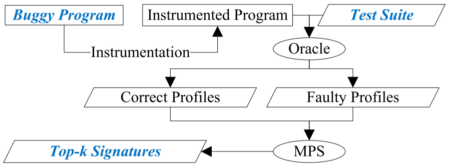

Figure 3: Overall Workflow to Bug Signature Identification

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 $p = null$ 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[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

ID	Input	Label	Predicates							
			2:true	4:false	7:true	9:true	12:false	14:false	11: f_ele == NULL?	16: i < n?
t_1	(null, 1)	$+$	●							
t_2	([], 1)	$+$		●			●			
t_3	([1], 2)	$+$		●	●	●		●	●	
t_4	([1, 2, 3], 3)	$+$		●	●	●		●		●
t_5	([1], 3)	$-$		●	●	●			●	●

Table 2: An Example Database Constructed from Table 1

ID	Transaction
t_1	$(\{2\}, +)$
t_2	$(\{4, 12\}, +)$
t_3	$(\{4, 7, 9, 14, 11\}, +)$
t_4	$(\{4, 7, 9, 14, 16\}, +)$
t_5	$(\{4, 7, 9, 11, 16\}, -)$

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 minimality 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 $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of distinct items, $\mathcal{C} = \{+, -\}$ be the set of the positive and negative class labels, and $\mathcal{D}$ be a database consisting of n transactions $\{(T_1, c_1), \cdots, (T_i, c_i), \cdots, (T_n, c_n)\}$, where $T_i \subseteq \mathcal{I}$ and $c_i \in \mathcal{C}$. In the context of debugging, $\mathcal{I}$ 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 $\mathcal{I}$.

Table 2 shows an example dataset constructed from the profiles of Table 1. $\mathcal{I}$ 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_ele == NULL).

We define two classification functions $+$ and $-$ for a set of transactions S,

$$S^+ = \{(T, c) \in S | c = +\} \text{ and } S^- = \{(T, c) \in S | c = -\}$$

For example, $\mathcal{D}^+$ or $\mathcal{D}^-$ denotes all the positive or negative transactions in $\mathcal{D}$ respectively. For an itemset (or pattern) $P \subseteq \mathcal{I}$, we define $tx : 2^{\mathcal{I}} \rightarrow 2^{\mathcal{D}}$, returning all transactions in $\mathcal{D}$ containing pattern P.

$$tx(P) = \{(T, c) \in \mathcal{D} | P \subseteq T\}$$

The *support* of P is defined as the number of transactions containing P, i.e. $sup(P) = |tx(P)|$; moreover, $sup^+(P) = |tx(P)^+|$ and $sup^-(P) = |tx(P)^-|$. The support of itemsets satisfies the following property stated in [2].

PROPERTY 1 (**APRIORI**). *Given a pattern $P \subseteq \mathcal{I}$, $\forall P' \subseteq \mathcal{I}$, if $P' \supset P$, then $tx(P') \subseteq tx(P)$, and further*

$$sup^+(P') \leq sup^+(P) \text{ and } sup^-(P') \leq sup^-(P)$$

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 $sup(\{4, 7\}) < sup(\{4\})$.

DEFINITION 1 (**EQUIVALENCE RELATION**). *Given an itemset database $\mathcal{D}$ defined over a set of items $\mathcal{I}$, the function $tx : 2^{\mathcal{I}} \rightarrow 2^{\mathcal{D}}$ induces an equivalence relation $\sim_{\mathcal{D}}$ on $2^{\mathcal{I}}$ such that for all itemset patterns $P_1, P_2 \in 2^{\mathcal{I}}$, $P_1 \sim_{\mathcal{D}} P_2$ if and only if $tx(P_1) = tx(P_2)$. Furthermore, the equivalence class $[P]$ of a pattern P is defined as $\{P' \subseteq \mathcal{I} | tx(P') = tx(P)\}$.*

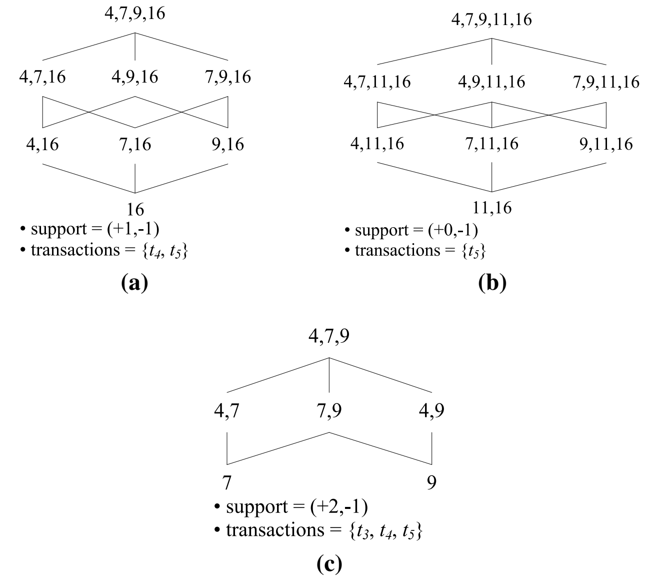

Figure 4: Three Equivalence Classes of Table 2

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 $P' \subset P$, $sup(P) < sup(P')$.*

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 $sup(\emptyset) = 5$ greater than $sup(\{16\})$. 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.

3. PROBLEM FORMULATION

3.1 Bug Signature

An interesting observation is that in Figure 4b, the generator $\{11, 16\}$ appears only in (all) the faulty profiles. This generator is the signature we are aiming to identify, as it is highly correlated to faulty executions. Furthermore, this generator carries as much predictive information as any supersets in the same equivalence class, as they all account for the same set of execution profiles. For example, the supersets $\{4, 11, 16\}$, $\{7, 11, 16\}$ and $\{9, 11, 16\}$ are all observed in the only faulty profile t_5. From the viewpoint of program semantics, predicate (4:false), (7:true) or (9:true) is redundant, as they are the necessary condition for execution to trigger the bug – statement 10. In other words, given the signature $\{11, 16\}$, we can infer that predicates 4, 7, 9 are also observed true in the same set of profiles as the signature. In the following, we formulate the concept of bug signatures based on generators.

DEFINITION 2 (**BUG SIGNATURE**). *Given a labeled itemset database, constructed from correct and faulty profiles of predicates, a bug signature is the set of generators of an equivalence class induced by the corresponding function tx.*

A bug signature is of the form $\{g_1, \cdots, g_n\}$, where g_i = $\{p_{i_1}, \ldots, p_{i_{m_i}}\}$ for each i is a generator. It can be perceived as a representative of an equivalence class. Its relationship with faulty executions is captured by the underlying tx function. The assumption we make of the bug signature is that: *The higher an equivalence class is positively correlated with faulty executions, the more probable its bug signature is related to the cause and/or effects of the bug.*

For example, the equivalence class in Figure 4c provides a bug signature $s_1 = \{\{7\}, \{9\}\}$ correlated with transaction t_3, t_4 and t_5. It means that if an execution has taken either the branch (7:true) *or* (9:true), it's probable that the execution is faulty. Another equivalence class in Figure 4b provides a bug signature $s_2 = \{\{11, 16\}\}$ correlated only with the (faulty) transaction t_5. It means that if an execution evaluates both predicates (11:f_ele == NULL) *and* (16:i < n) to true, it's probable that the execution is faulty. Here, we would expect that s_2 captures the cause and effects of bug with higher possibility than s_1, since s_2 occurs only in (all) the faulty executions. In other words, due to its association with faulty and not correct executions, s_2 is better than s_1 in discriminating faulty executions from correct ones.

In the following section, we introduce a metric to measure the discriminative power of a signature.

3.2 Discriminative Significance

The basic unit of our signatures is itemset generator, so we start by discussing how discriminative significance of a pattern is computed. A pattern is deemed discriminative if it can be used to distinguish one class of transactions from the other. The significance of a discriminative pattern is typically measured by the notion of *information gain* (IG) [22]. Let $\mathcal{D}$ be a database of class-labeled transactions. Given a pattern P, its information gain is high if it appears frequently in one class of transactions, whereas rarely in the other. Let $p = sup^+(P)$, $n = sup^-(P)$, then the information gain of pattern P can be defined in Equation 1,

$$IG(p, n) = H(|\mathcal{D}^+|, |\mathcal{D}^-|) - \frac{p+n}{|\mathcal{D}|} \times H(p, n) - \frac{|\mathcal{D}| - (p+n)}{|\mathcal{D}|} \times H(|\mathcal{D}^+| - p, |\mathcal{D}^-| - n) \qquad (1)$$

where

$$H(a, b) = -\frac{a}{a+b} \times \log_2(\frac{a}{a+b}) - \frac{b}{a+b} \times \log_2(\frac{b}{a+b})$$

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:

$$DS(p, n) = \begin{cases} IG(p, n) & \text{if } \frac{n}{|\mathcal{D}^-|} > \frac{p}{|\mathcal{D}^+|} \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

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 {{(11:f_ele==NULL), (16:i<n)}} is the best, for it has the highest discriminative significance score.

Table 3: Signatures for Profiles in Table 1

Rank	Support	Signatures	DS
1	$(+0, -1)$	$\{\{11, 16\}\}$	0.721928
2	$(+1, -1)$	$\{\{11\}\}$	0.321928
3	$(+1, -1)$	$\{\{16\}\}$	0.321928
4	$(+2, -1)$	$\{\{7\}, \{9\}\}$	0.170951
5	$(+3, -1)$	$\{\{4\}\}$	0.072906

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 $GTree_{db}^{prefix}$, where *prefix* is an itemset prefixing all the items in the Gr-tree. For the original database $\mathcal{D}$, *prefix* is $\emptyset$. Figure 5 shows the Gr-tree of database db in Table 2, $GTree_{db}^{\emptyset}$. 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.

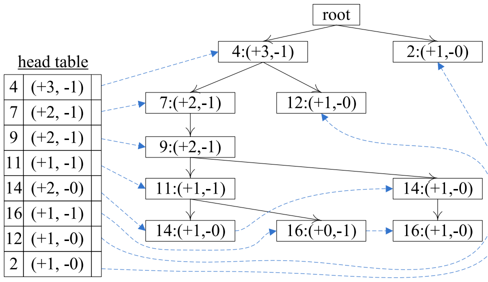

Figure 5: Gr-tree of the Database of Table 2 with *prefix* $= \emptyset$

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* (*ie.*, 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:$(+p, -n)$ represents an itemset pattern comprising of the items in the path, supported by p positive and n negative transactions. For example, the path $\langle root$, 4:$(+3, -1)$, 7:$(+2, -1)$, 9:$(+2, -1)$, 11:$(+1, -1)$, 16:$(+0, -1)\rangle$ 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 $GTree_{db}^{px}$, let $a_1, \cdots, a_n$ be items in its head table. Then the conditional database of $a_i (1 \leq i \leq n)$ is denoted by $CD|_{GTree_{db}^{px}}^{px \cup \{a_i\}}$, as the set of path segments exclusively between the root and a_i for all paths containing a_i.*

A conditional database $CD|_{GTree_{db}^{px}}^{px \cup \{a_i\}}$ 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 $GTree_{db}^{\emptyset}$ in Figure 5 with respect to item 16, $CD|_{GTree_{db}^{\emptyset}}^{\emptyset \cup \{16\}}$. In order to construct this conditional database from $GTree_{db}^{\emptyset}$, we first extract all path segments between $root$ and nodes 16:

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

ID	Transaction
t_4	$(\{4, 7, 9, 14\}, +)$
t_5	$(\{4, 7, 9, 11\}, -)$

1. $\langle root$, 4:(+3,-1), 7:(+2,-1), 9:(+2,-1), 11:(+1,-1), 16:(+0,-1)$\rangle$

2. $\langle root$, 4:(+3,-1), 7:(+2,-1), 9:(+2,-1), 14:(+1,-0), 16:(+1,-0)$\rangle$

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 $CD|_{GTree_{db}^{\emptyset}}^{\emptyset \cup \{16\}}$. Then we can build its Gr-tree, $GTree|_{cd}^{\emptyset \cup \{16\}}$ where $cd = CD|_{GTree_{db}^{\emptyset}}^{\emptyset \cup \{16\}}$ shown in the figure below.

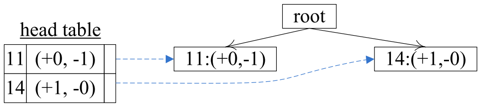

Figure 6: Gr-tree of the Conditional Database of Table 4 with *prefix* $= \{16\}$

Note that items 4, 7, 9 are removed from Gr-tree, as they have full support in the conditional database $CD|_{GTree_{db}^{\emptyset}}^{\emptyset \cup \{16\}}$.

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 $gens = \{g_i\}_i^m$ 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: $MineSignatures(\mathcal{D}, k, neg\_sup, size\_limit)$
Input: $\mathcal{D}$, 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
Input: $size\_limit$, upper size limit of generators in signatures
Output: $RS$, a list containing top-$k$ discriminative signatures
1 $GS := []$;
2 $MineRec(GTree_{\mathcal{D}}^{\emptyset}, k, neg\_sup, size\_limit, GS)$;
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 $\mathcal{D}$ we create its Gr-tree with empty prefix, and pass $GTree_{\mathcal{D}}^{\emptyset}$ 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 stops, 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.

Algorithm 2: $MineRec(tree, k, neg\_sup, size\_limit, GS)$

```
   Input: $tree$: $GTree^{px}_{db}$ with prefix $px$, constructed from $db$
   Input: $k$, $neg\_sup$, $size\_limit$: the same as those in Algorithm 1
   Input: $GS$: a list to store mined top-$k$ discriminative generators
   Output: mined generators are stored in $GS$
 1 if $tree$ is empty then return;
 2 foreach item $i$ in the head table of $tree$ do
 3     $px' := px \cup \{i\}$;
 4     $UpdateResult(GS, k, sup^{+}(px'), sup^{-}(px'), px')$;
 5 if $|px| + 1 = size\_limit$ then return;
 6 if $tree$ is a single path then return;
 7 foreach item $i$ in the head table of $tree$ do
 8     $px' := px \cup \{i\}$;
 9     let $db'$ denote the conditional database $CD|^{px'}_{tree}$;
10     remove any item $a$ from $db'$ if $sup^{-}(a) < neg\_sup$;
11     let $tree'$ denote the Gr-tree $GTree^{px'}_{db'}$ of $db'$;
12     if $|GS| = k \wedge UpperBound(tree') < MinDS(GS)$ then
13         continue;              // Branch and Bound Here!
14     $MineRec(tree', k, neg\_sup, size\_limit, GS)$;
15 Procedure $UpdateResult(GS, k, p, n, gen)$:
16 if $\exists e : e = ((p, n), gens) \wedge e \in GS$ then
17     $GS := GS \backslash \{e\} \cup \{((p, n), gens \cup \{gen\})\}$;
18 else
19     $GS := GS \cup \{((p, n), \{gen\})\}$;
20     if $|GS| > k$ then
21         choose $e \in GS$ such that
           $e = ((p', n'), gens') \wedge MinDS(GS) = DS(p', n')$;
22         $GS := GS \backslash \{e\}$;
```

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 $ux(P) \subseteq tx(P)$ which contains all super patterns of interest $P' \supseteq P$, referred to as *unavoidable* transactions.

$$ux(P) = \bigcap_{P' \supseteq P} tx(P')$$

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[2]:

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:*

$$UB(P) = \begin{cases} IG(|ux(P)^{+}|, sup^{-}(P)) & \textit{if } \frac{sup^{-}(P)}{|D^{-}|} > \frac{|ux(P)^{+}|}{|D^{+}|} \\ 0 & \textit{otherwise} \end{cases}$$

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

[2] The proof is presented in Section 9.

upper bound of all these patterns (i.e. $UpperBound(tree)$ invoked at lines 12 in Algorithm 2) can be computed as follows. We iterate every path of $tree$ from *root* to leaf nodes, until we find a path $\langle root, a{:}(+m, -n), \cdots, ln{:}(+p, -q)\rangle$ containing all items in the head table. The transactions containing $\{a, \cdots, ln\}$ 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 ln, namely

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

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

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., *print_tokens*, Unix utilities and *space* interpreter) on a PC with Intel Core 2 Quad CPU 3.0GHz and 8GB memory. The *print_tokens* is a subject in Siemens benchmark which was developed to test the testing coverage strategies [13]. The programs *grep*, *gzip*, *sed* are Unix utility programs of moderate size, and the last subject *space* is an interpreter for an array definition language. The sizes of the programs range from 726 to 14,427.

Table 5: Benchmark Statistics

Subject	LoC	#Test Cases	#Faults
print_tokens	726	4130	7
grep	10,068	199	12
gzip	5,680	214	16
sed	14,427	360	9
space	6,199	13585	31
total			75

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., *mps-inter*, *mps-intra*, *leap-inter* and *leap-intra*, and use them to identify the top-1 bug signature. For MPS, we set the parameters $neg\_sup = 0.5$[4] and $size\_limit = 2$. 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 measure the quality of mined bug signatures (in assisting in locating bug in a program) by computing the proximity of the signature to the actual bug. 3) We present the statistics of runtime.

The artifact (including source code, tools and supportive data) used in our case studies has been successfully evaluated by the ESEC/FSE artifact evaluation committee and found to meet expectations. It is available at `www.comp.nus.edu.sg/~specmine/suncn/mps-artifacts`.

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 ($IG_{MPS} - IG_{LEAP}$).

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 in [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: Improvement in Information Gain of MPS

	mean	median	p
mps-inter v.s. *leap-inter*	0.25	0.13	< 0.0001
mps-inter v.s. *leap-intra*	0.26	0.19	< 0.0001
mps-intra v.s. *leap-inter*	0.24	0.14	< 0.0001
mps-inter v.s. *leap-intra*	0.25	0.19	< 0.0001

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 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

[3] `http://research.cs.wisc.edu/cbi/`

[4] To simplicity, we use relative support instead of absolute support.

Table 7: Proximity Results

subject	median						mean					
	LEAP		MPS				LEAP		MPS			
	inter	intra	inter	impr.	intra	impr.	inter	intra	inter	impr.	intra	impr.
print_tokens	0.718	0.859	0.959	11.7%	0.959	11.7%	0.645	0.809	0.897	10.9%	0.910	12.4%
grep	0.505	0.358	0.912	80.5%	0.889	76.0%	0.390	0.335	0.780	100.2%	0.791	102.9%
gzip	0.003	0.688	0.898	30.5%	0.891	29.5%	0.224	0.503	0.831	65.1%	0.812	61.5%
sed	0.648	0.692	0.919	32.9%	0.952	37.6%	0.518	0.635	0.911	43.4%	0.936	47.4%
space	0.000	0.000	0.992	–	0.996	–	0.000	0.053	0.856	1518.0%	0.878	1559.6%

distance $d(n)$ between any node n in the signature (S) and the node b for the actual bug. Different from the approach in [23, 9], in which the distance is defined as the number of edges in the shortest *directed* path connecting a node n in S and the bug b, we relax the directional constraint of path to *undirected*, as the information carried in predicates enables programmers to reason bugs in an undirected way. For example, the following code snippet is extracted from *print_tokens*, and the bug is the misplacement of *case* 32. MPS outputs a signature containing a predicate (*cu_state* == 32) right after the statement (*cu_state* = *next_st*). There is no directed dependency path from *case* 32 to the predicate location. However, with this predicate, we know that *next_st* is also 32, and this variable can directly lead the execution to reach the bug point. Thus the undirected path between *cu_state* and *case* 32 via *next_st* provides a good clue for debugging.

```
next_st = ...;
switch(next_st) {
  ...
  case 32: ... // bug is "case 32"
  ...
}
cu_state = next_st;
// a signature here: cu_state == 32
```

After getting the shortest distance, we compute

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

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:

$$\mathcal{X} = 1 - \frac{|N|}{|PDG|}$$

We used CODESURFER to compute the PDG of the buggy program, and compute $\mathcal{X}$ 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.,

$$\frac{\textit{mps-inter} - \textit{max}(\textit{leap-inter}, \textit{leap-intra})}{\textit{max}(\textit{leap-inter}, \textit{leap-intra})}$$

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

```
1   static int inchar() {
2     if (prog.cur) {
3       //bug: '<=' should be '<'
4       if (prog.cur <= prog.end)
5         ch = *prog.cur++;
6     } else if (prog.file) {
7       if (!feof(prog.file))
8         ch = getc(prog.file);
9     }
10    if (ch == '\n')
11      ++cur_input.line;
12    return ch;
13  }
14  int main(int argc, char* argv[]) {
15    ......
16    while ((opt = getopt_long(...)) != EOF) {
17      switch(opt) {
18      case 'e': compile_string(...); break;
19      case 'f': compile_file(...); break;
20      ......
```

Figure 7: A Bug in *sed*

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

Table 8: Runtime Statistics (in seconds)

subject	mps-inter	leap-inter	impr.	mps-intra	leap-intra	impr.
print_tokens	50.32	76.38	51.79%	11.45	27.51	140.46%
grep	18.79	77.28	311.18%	0.30	38.75	12800%
gzip	131.44	31.71	-75.88%	1.55	28.29	1722.4%
sed	82.30	104.94	27.51%	5.66	51.31	806.30%
space	781.40	–	–	6.93	208.37	2900.1%

```
1  bool Two_of_Three_Reports_Valid; //global
2  int Other_RAC; //global
3  int alt_sep_test() {
4    ......
5    bool tcas_equipped=(Other_Capacity==1);
6    bool intent_not_known=
7      Two_of_Three_Reports_Valid ||
8      (Other_RAC==0);//bug: '||' should be '&&'
9    ......
10   if(enabled &&
11       ((tcas_equipped && intent_not_known)
12         || !tcas_equipped)) {
13     ......
14     alt_sep = 1;
15     ......
16 }
17 void main(int argc, char* argv[]) {
18   ......
19   Two_of_Three_Reports_Valid = atoi(argv[3]);
20   Other_RAC = atoi(argv[10]);
21   ......
22 }
```

Figure 8: A Bug in Version 3 of *tcas*

bug happens when users provide a *sed* command via the command option '-e' and (*prog.cur* == *prog.end*). The program is safe if the command is stored in a file specified with the command option '-f'. MPS outputs a signature including two predicates for this bug:

(*ch* < 1) at line 5, and (*opt* == 'e') 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 $DS = 0.28$, whereas the LEAP signature appears in 149 correct and 13 faulty executions with $DS = 0.01$.

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 &&, shown in Figure 8. The bug manifests in the following two scenarios, when the boolean variable *intent_not_known* is incorrectly assigned with value *true* instead of *false*:

1) *Two_of_Three_Reports_Valid* == 0 and *Other_RAC* == 0
2) *Two_of_Three_Reports_Valid* == 1 and *Other_RAC* $\neq$ 0

MPS outputs a signature containing two predicates:

1) (*Other_RAC* $\geq$ *Two_of_Three_Reports_Valid*) after line 20
2) (*alt_sep* == *tcas_equipped*) after line 14

The following is our process to diagnose the bug. As *alt_sep* has been assigned with 1 at line 14, *tcas_equipped* must be 1 (as 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 $DS = 0.08$.

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 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. Renieris 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]. As pointed out in their paper, they only use single suspicious predicates in each function to guide the test generation as identifying multiple predicates takes longer time. Our mining technique (e.g., *mps-intra*) can complement theirs by overcoming this constraint.

Discriminative Pattern Mining. Hong et al. mine discriminative 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 functionality of our signatures is extended to enable developers to diagnose a class of bugs, the manifestation of which does not trigger any deviation in control-flow transitions from correct executions, and thus cannot be detected by control-flow-based signatures. Although the information presented in the profiles has increased, we manage to produce signatures of smaller sizes through the technique of itemset generator mining.

Moving forward, we will explore various opportunities for optimizing the mining algorithm. We are in the progress of developing a debugging environment that assists programmers in inferring the cause of bugs from the data predicates present in signatures. In future, we plan to conduct human studies to investigate the effectiveness of our approach in debugging large real-world programs. We also plan to enhance the algorithm to assist in discovery of multiple bugs present in the execution profiles.

8. ACKNOWLEDGMENT

We are grateful to Yan Han Pang, Theong Siang Oo, Kheng Meng Yeo and Thao Nguyen at National University of Singapore for their assistance in conducting the case studies. We thank the anonymous reviewers for their valuable comments and evaluation of the paper and the associated artifacts. Our appreciation also goes to Hong Cheng at Chinese University of Hong Kong for providing the LEAP tool, and Ben Liblit at University of Wisconsin-Madison for making CBI instrumentor publicly available. This work is supported by a research grant R-252-000-484-112.

9. SUPPLEMENTARY INFORMATION

The following is the proof of Theorem 1.

PROOF. Let $\mathcal{D}$ be an edge label transaction database, P be a pattern, $p = sup^{+}(P)$ and $n = sup^{-}(P)$.
The partial derivative of IG $w.r.t$ the positive support p:

$$\frac{\partial IG}{\partial p} = \frac{1}{|\mathcal{D}|} \log_2 \frac{p(|\mathcal{D}| - p - n)}{(p+n)(|\mathcal{D}^{+}| - p)}$$

The partial derivative of IG $w.r.t$ the negative support n:

$$\frac{\partial IG}{\partial n} = \frac{1}{|\mathcal{D}|} \log_2 \frac{n(|\mathcal{D}| - p - n)}{(p+n)(|\mathcal{D}^{-}| - n)}$$

Let P' be a qualified super pattern of P, $p' = sup^{+}(P'), n' = sup^{-}(P')$, then $|ux(P)^{+}| \leq p' \Rightarrow \frac{|ux(P)^{+}|}{|D^{+}|} \leq \frac{p'}{|D^{+}|}$ based on the definition of unavoidable transactions.

1. If $\frac{n'}{|\mathcal{D}^{-}|} \leq \frac{|ux(P)^{+}|}{|D^{+}|}$, then $\frac{n'}{|\mathcal{D}^{-}|} \leq \frac{p'}{|\mathcal{D}^{+}|}$, thus $DS(p', n') = 0$.
2. If $\frac{n'}{|\mathcal{D}^{-}|} > \frac{|ux(P)^{+}|}{|D^{+}|}$, we discuss the following two cases. First, if $\frac{n'}{|\mathcal{D}^{-}|} \leq \frac{p'}{|\mathcal{D}^{+}|}$, then $DS(p', n') = 0$. Second, if $\frac{n'}{|\mathcal{D}^{-}|} > \frac{p'}{|\mathcal{D}^{+}|}$, then $\frac{\partial IG}{\partial p} < 0$ and $\frac{\partial IG}{\partial n} > 0$ thus IG is monotonically decreasing to p, and monotonically increasing to n. Then $DS(p', n') = IG(p', n') \leq IG(|ux(P)^{+}|, n') \leq IG(|ux(P)^{+}|, sup^{-}(P))$.

So the upper bound of DS is correct. □

10. REFERENCES

[1] R. Abreu. *Spectrum-Based Fault Localization in Embedded Software.* PhD thesis, Delft University of Technology, 2009.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB*, 1994.

[3] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical Debugging Using Compound Boolean Predicates. In *ISSTA*, 2007.

[4] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal Inference for Statistical Fault Localization. In *ISSTA*, 2010.

[5] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying Bug Signatures Using Discriminative Graph Mining. In *ISSTA*, 2009.

[6] H. Cheng, X. Yan, J. Han, and C.-W. Hsu. Discriminative Frequent Pattern Analysis for Effective Classification. In *ICDE*, 2007.

[7] H. Cheng, X. Yan, J. Han, and P. S. Yu. Direct Discriminative Pattern Mining for Effective Classification. In *ICDE*, 2008.

[8] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *ICSE*, 2009.

[9] H. Cleve and A. Zeller. Locating Causes of Program Failures. In *ICSE*, 2005.

[10] C.Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. SOBER: Statistical Model-based Bug Localization. In *FSE*, 2005.

[11] R. Gore and P. F. Reynolds, Jr. Reducing Confounding Bias in Predicate-level Statistical Debugging Metrics. In *ICSE*, 2012.

[12] H. Hsu, J. A. Jones, and A. Orso. RAPID: Identifying Bug Signatures to Support Debugging Activities. In *ASE*, 2008.

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *ICSE*, 1994.

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

[15] J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum Description Length Principle: Generators Are Preferable to Closed Patterns. In *AAAI*, 2006.

[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation via Remote Program Sampling. In *PLDI*, 2003.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *PLDI*, 2005.

[18] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. In *KDD*, 2009.

[19] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive Evaluation of Association Measures for Fault Localization. In *ICSM*, pages 1–10, 2010.

[20] S. Nijssen, T. Guns, and L. Raedt. Correlated Itemset Mining in ROC Space: A Constraint Programming Approach. In *KDD*, 2009.

[21] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *ISSTA*, 2011.

[22] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[23] M. Renieris and S. Reiss. Fault Localization With Nearest Neighbor Queries. In *ASE*, 2003.

[24] J. Rossler, G. Fraser, A. Zeller, and A. Orso. Isolating Failure Causes through Test Case Generation. In *ISSTA*, pages 309–319, 2012.

[25] C. Sun, J. Du, N. Chen, S.-C. Khoo, and Y. Yang. Mining Explicit Rules for Software Process Evaluation. In *ICSSP*, pages 118–125, 2013.

[26] J. Wang and J. Han. BIDE: Efficient Mining of Frequent Closed Sequences. In *ICDE*, 2004.

[27] X. Yan, H. Cheng, J. Han, and P. Yu. Mining Significant Graph Patterns by Leap Search. In *SIGMOD*, 2008.

[28] X. Zhang, N. Gupta, and R. Gupta. Locating Faults through Automated Predicate Switching. In *ICSE*, 2006.