Efficient Predicated Bug Signature Mining via Hierarchical Instrumentation

Zhiqiang Zuo
School of Computing
National University of Singapore
zhiqiangzuo@nus.edu.sg

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

Chengnian Sun
Department of Computer Science
University of California, Davis
cnsun@ucdavis.edu

ABSTRACT

Debugging is known to be a notoriously painstaking and time-consuming task. An essential and yet expensive process in debugging is bug isolation. As one major family of automatic bug isolation, statistical bug isolation approaches have been well studied in the past decade. A recent advancement in this area is the introduction of *bug signature* that provides contextual information to assist in debugging and several bug signature mining approaches have been reported. All these approaches instrument the entire buggy program to produce profiles for debugging. Consequently, they often incur hefty instrumentation and analysis cost. However, as in fact major part of the program code is error-free, full-scale program instrumentation is wasteful and unnecessary. In this paper, we devise a novel *hierarchical instrumentation* (HI) technique to perform selective instrumentation so as to enhance the efficiency of statistical debugging. We employ HI technique to predicated bug signature mining (called MPS) recently developed and propose an approach called HIMPS. The empirical study reveals that our technique can achieve around 40% to 60% saving in disk storage usage, time and memory consumption, and performs especially well on large programs. It greatly improves the efficiency of bug signature mining, making a step forward to painless debugging.

Categories and Subject Descriptors

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

General Terms

Experimentation, Reliability

Keywords

automated debugging, bug signature mining, hierarchical instrumentation

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
ACM 978-1-4503-2645-2/14/07
http://dx.doi.org/10.1145/2610384.2610400

1. INTRODUCTION

As is well known, debugging is a notoriously painstaking and time-consuming task. As an essential and yet expensive process in debugging, bug isolation (or fault localization) aims to isolate or locate program bugs. A considerable number of automatic bug isolation approaches have been studied in the past decade. Statistical bug isolation [1, 2, 3, 4, 5, 6, 7] is one major family of these automatic approaches. They collect two groups (i.e., failing and passing) of executions and apply the statistical techniques to pinpoint the discriminative elements as the potential failure causes.

Recently, Parnin and Orso [8] claimed that *perfect bug understanding* does not hold. It is difficult in practice to understand the bug by examining a single buggy statement. More contextual information where the bug occurs is likely to provide useful clue for identifying, understanding and correcting bugs. Hsu et al. [9] coined the term *bug signature*. Instead of a single suspicious element (statement or predicate) isolated by automated bug isolation, bug signature comprises multiple elements providing the bug *context* information. They adopted sequence mining algorithm to discover longest sequences in a set of failing executions as bug signatures. Subsequently, Cheng et al. [10] identified bug signatures using discriminative graph mining. They mined discriminative control flow graph patterns as bug signatures from both passing and failing executions. Since only control flow transitions are considered in [10], bugs not causing any deviation in control flow transitions cannot be identified. To enhance the predictive power of bug signatures, Sun and Khoo [11] proposed *predicated bug signature mining*, where both data predicates and control flow information are utilized. They devised a discriminative itemset generator mining technique to discover succinct predicated bug signatures.

Motivation. Both statistical bug isolation [1, 2, 3, 5, 6, 7] and bug signature identification approaches [10, 11] in essence analyze failing and passing executions to identify discriminative elements as the bug cause or bug signature. They assume that *every program element* can be a candidate for bug identification, and thus instrument *the entire program* before performing bug discovery task. Such full-scale program instrumentation incurs hefty cost in terms of disk storage space consumption, CPU time and memory usage, etc., not just during instrumentation but also the analysis thereafter. However, in fact, most part of the program code works properly, and *only small portions of a program are relevant to a given bug* [12]. As stated in [3], the majority of the predicates tracked (often 98-99%) are irrelevant to program failures. This motivates us to develop a selective instrumen-

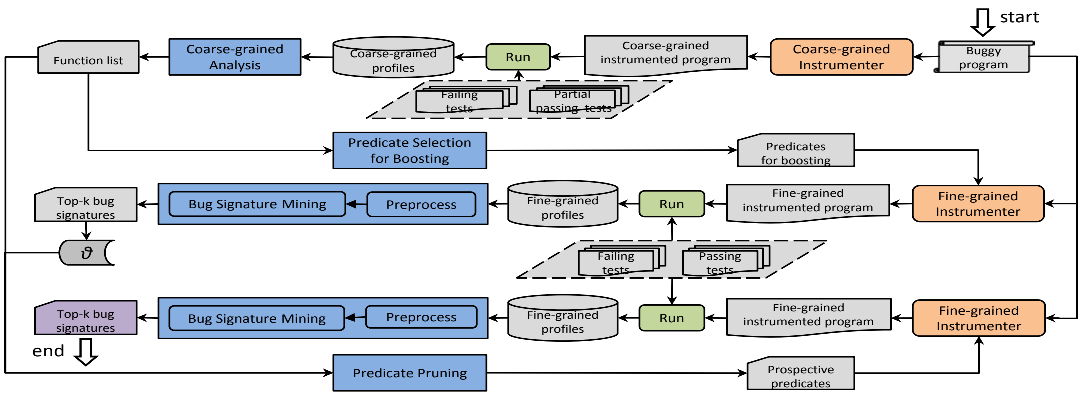


Figure 1: Workflow of predicated bug signature mining via hierarchical instrumentation

tation technique such that only the program elements which are highly correlated to the failure are instrumented, thus improving the efficiency of statistical debugging.

Approach. To this end, we devise a novel *Hierarchical Instrumentation* (HI) technique to perform selective instrumentation. Our work is based on the insight that *information collected and measured by instrumenting composite syntactic constructs (e.g., functions) can be used to guide the selection of program elements (e.g., predicates) for subsequent instrumentation.* We term the former instrumentation *coarse-grained* whereas the latter instrumentation *fine-grained*. Briefly, a lightweight coarse-grained instrumentation is first performed to obtain the execution information of coarse-grained elements (e.g., functions). Through these coarse-grained information, we can safely and effectively prune away instrumentation (and subsequent analysis) of a substantial number of fine-grained elements (e.g., predicates).

In this paper, we demonstrate the effectiveness of HI technique by applying it to predicated bug signature mining described in [11]. We propose an efficient predicated bug signature mining approach via HI, called HIMPS, whose workflow is demonstrated by Figure 1.

Specifically, HIMPS comprises two phases: one-pass coarse-grained phase followed by two-pass fine-grained phase. At coarse-grained phase, we instrument all the function entries (called coarse-grained elements) of the program and run the instrumented program over a small portion of passing test cases and all the failing test cases. We then capture the execution information of these function entries, which is then used in the fine-grained phase to guide the selective fine-grained instrumentation. The fine-grained phase comprises two passes, namely *boosting* and *pruning*. The boosting pass instruments and analyzes a selected subset of predicates (called fine-grained elements), and computes a fine-grained suspiciousness threshold from it. This threshold is then exploited in the pruning pass, for pruning away unnecessary predicates and returning a set of prospective predicates for fine-grained instrumentation. Only these prospective predicates will be instrumented during the pruning pass which finally returns the top-ranked bug signatures as output for our entire analysis.

We conduct experiments to compare HIMPS against the MPS system developed in [11]. The experiments validate that HI technique can greatly improve the time and space efficiency of MPS without jeopardizing the effectiveness of mining for top-ranked signatures: it overall saves near 40% of execution time and more than 50% of disk storage space for profile collection, and only takes less than 60% of time and peak memory consumption for preprocessing and mining together. It performs especially well on large programs.

Contribution. Our contributions are as follows.

- We devise a novel *hierarchical instrumentation* technique to conduct selective instrumentation. To the best of our knowledge, this is the first which exploits the coarse-grained execution information to guide the fine-grained instrumentation for automated debugging.
- We propose HIMPS, an efficient predicated bug signature mining via hierarchical instrumentation, and demonstrate empirically that it significantly outperforms the existing MPS in terms of both space and time efficiency.
- We provide evidence to support that the HI technique is feasible, and rigorously prove that our HIMPS approach is safe in discovery of top-k bug signatures.

Outline. The remainder of this paper is organized as follows. Section 2 introduces some background about predicated bug signature mining, followed by the detailed presentation of our approach in Section 3. We discuss the experimental evaluation of our work in Section 4. Section 5 gives the literature. Finally, Section 6 concludes.

2. BACKGROUND

We explain some of the steps used in statistical debugging approaches through the lens of predicated bug signature discovery proposed by Sun and Khoo [11]. Specifically, we discuss in some details how a program is instrumented to produce predicates during execution, the metrics used in assessing the suspiciousness of signatures – in the form of itemset, and how preprocessing is done to reduce the size of database for signature mining.

2.1 Predicated Bug Signature

An instrumentation scheme widely used in the statistical debugging community was developed by Liblit et al. [3]. It was also adopted by Sun and Khoo [11] in their design of predicated bug signature discovery. Here, a program is instrumented to collect the runtime values of *predicates* at particular program points. Each program point to be instrumented is called *instrumentation site*. At each instrumentation site, several *predicates* are tracked. There are four categories of instrumentation sites considered:

- **Branches**: For each conditional, two predicates are tracked to indicate whether the *true* or *false* branch is taken at runtime.
- **Returns**: At each scalar-returning call site, six predicates are created to capture whether the return value r is ever $> 0, \geq 0, < 0, \leq 0, = 0$, or $\neq 0$.
- **Scalar-pairs**: At each assignment of a scalar value, six predicates are considered: $x <, \leq, >, \geq, =, \neq y_i$ (or c_j), where x is the assigned value, y_i and c_j represent one of the other same-typed in-scope variables and one of the constant-valued integer expressions seen in the program, respectively.
- **Float-kinds**: At each assignment of a floating point value, nine predicates are recorded to check if the assigned value is: *-Inf, negative and normalized, negative and denormalized, -0, NaN, +0, positive and denormalized, positive and normalized, +Inf.*

A profile is obtained for each run of the instrumented program. It consists of a set of predicate counts which records the number of times each predicate is evaluated to true during the run. In [11], only those predicates whose counts are equal to or bigger than 1 (i.e., the predicate is evaluated to true at least once), are retained. Each profile is thus regarded as a set of items, each of which is a predicate evaluated to true at least once during execution. In addition, each profile is labeled as passing or failing according to the oracle. All the profiles constitute a database of labeled itemset transactions, each corresponding to one profile.

Formally, let $\mathcal{I} = \{e_1, e_2, \ldots, e_m\}$ be a set of items, $\mathcal{C} = \{+, -\}$ be the set of class labels, $\mathcal{D}$ be a class-labeled itemset database constituting n transactions, i.e., $\mathcal{D} = \{(T_1, c_1), \ldots, (T_n, c_n)\}$ where $\forall i \in [1, n], T_i \subseteq \mathcal{I} \wedge c_i \in \mathcal{C}$. As discussed above, in the context of predicated bug signature mining, $\mathcal{I}$ corresponds to the set of all the instrumented predicates. Each transaction T_i in $\mathcal{D}$ corresponds to a profile consisting of predicates evaluated to true during execution – a subset of $\mathcal{I}$. Each profile is generated by running the instrumented program using one test case. The class label (+) identifies the passing profile (i.e., correct execution), whereas (-) labels the failing profile (i.e., faulty execution). We call the corresponding transactions *positive* and *negative* transactions, respectively.

In [11], a bug signature is a set of predicate itemsets, which are observed together frequently in the failing executions but rarely in the passing ones, and thus regarded to be correlated to program failures. Given the class-labeled itemset database, predicated bug signature identification is formulated as a *discriminative itemset pattern mining* task. They mine the highly discriminative bug signatures based on the discriminative measure discussed later in Section 2.2.

2.2 Discriminative Significance

The discriminative significance of a pattern (itemset) is typically measured by the notion of *information gain* (IG) [13].

Let $\mathcal{D}$ be a class-labeled itemset database, $\mathcal{D}^+$ and $\mathcal{D}^-$ denote all the positive and negative transactions in $\mathcal{D}$, respectively. Given an itemset pattern P, the *support* of P *wrt.* an itemset database $\mathcal{D}$ is defined as the number of transactions in $\mathcal{D}$ containing P, i.e., $sup(P, \mathcal{D}) = |td(P, \mathcal{D})|$ where $td(P, \mathcal{D}) = \{(T, c) \in \mathcal{D} | P \subseteq T\}$. let $p = sup^+(P, \mathcal{D}) = |td(P, \mathcal{D}^+)|$ and $n = sup^-(P, \mathcal{D}) = |td(P, \mathcal{D}^-)|$ be the number of all the positive and negative transactions containing P, which are called *positive support* and *negative support* of P, respectively. The information gain of pattern P can be defined as follows:

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

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 [11], Sun and Khoo define the following *discriminative significance* measure based on IG:

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

Since all the itemsets within one bug signature possess the same positive and negative support, their DS values are also the same. Therefore, the DS value is used as the discriminative significance score of a signature.

2.3 Preprocessing and Bug Signature Mining

In [11], Sun and Khoo first performed preprocessing on the profiles to produce a dataset which is subsequently fed into the bug signature miner. To begin with, some unimportant or redundant predicates are filtered out in advance. Specifically, three filtering strategies are applied. Firstly, according to the definition of discriminative significance (Equation 2), all predicates such that $\frac{n}{|\mathcal{D}^-|} \leq \frac{p}{|\mathcal{D}^+|}$ are filtered as their DS values are always zero. Secondly, all predicates whose *Increase* [3] value not greater than zero are also filtered. Thirdly, all predicates with operators $\geq, \leq, \neq$ are filtered if they and their subsumed predicates (with operators $>, <, =$) are both true in the same set of profiles. For instance, consider a predicate $a \geq b$, if this predicate is true in all the same profiles where $a > b$ is true. That means $a \geq b$ does not capture additional execution information than its subsumed predicate $a > b$. The predicate $a \geq b$ is actually redundant and thus can be filtered. Note that preprocessing is essential as it constructs the database in a suitable format for the subsequent mining step; furthermore, it filters a great number of predicates so as to effectively reduce the scale of mining. However, it is also quite expensive especially if the profiles processed are of big size.

Sun and Khoo [11] devised the *discriminative itemset generator mining* algorithm. Given a predicate itemset database constructed from profiles through preprocessing, and the number of top discriminative signatures to mine k, the algorithm discovers the top-k discriminative bug signatures based on the *discriminative significance* measure discussed in Section 2.2. Specifically, they adopted a tree-based representation of the database as [14] and proposed a depth-first search algorithm over the pattern space while effectively pruning the search space in a branch and bound fashion. They provided two modes of signature mining: *inter-procedural* and *intra-procedural*. In the first mode, a bug signature is identified over the whole program. The items in a signature can span across multiple functions. In the latter mode, the mining is employed to each function separately, and the items in a signature must reside in the same function. Since the *inter-procedural* signature mining is much more expensive than the *intra-procedural*, we focus on improving the efficiency of the *inter-procedural* mode in this work. We refer the readers to [11] for the detailed mining algorithm.

3. APPROACH

Given a buggy program and two groups (failing and passing) of test cases, the objective of our approach is to efficiently mine the top-k bug signatures which are highly correlated to the bug as measured by the DS values. The essence of our approach is a *safe pruning* of predicates instrumented and mined, making the bug signature mining more efficient. Briefly, we first capture the execution information of *functions* by a lightweight coarse-grained (function-level) instrumentation and analysis phase. Subsequently such information about functions is exploited to safely prune away unnecessary predicates. Algorithm 1 gives the detailed predicated bug signature mining algorithm via HI involving two phases, namely coarse-grained and fine-grained phase.

Algorithm 1: Predicated Bug Signature Mining via Hierarchical Instrumentation

```
   // coarse-grained instrumentation and analysis
1  1 Instrument all function entries in the entire program;
   2 Run all failing and partial passing test cases to collect
     coarse-grained profiles $CP$;
   3 $list \leftarrow AnalyzeCoarseGrainedProfiles(CP)$;
   // fine-grained instrumentation and analysis
   // first pass:  threshold boosting
   4 $boost \leftarrow SelectPredicatesForBoosting(list, \gamma)$;
2  5 Instrument all predicates in $boost$;
   6 Run all failing and passing test cases to collect all the
     fine-grained profiles $BP$;
   7 $BD \leftarrow Preprocess(BP)$;
   8 $BS \leftarrow MineBugSignatures(BD, k)$;
   9 $\theta \leftarrow$ the top-$k$th $DS$ value of signatures;
   // second pass:  safe pruning
   10 $prospect \leftarrow PrunePredicates(list, \theta)$;
3  11 Instrument all predicates in $prospect - boost$;
   12 Run all failing and passing test cases to collect all the
     fine-grained profiles $PP$;
   13 $PD \leftarrow Preprocess(PP+BP)$;
   14 $PS \leftarrow MineBugSignatures(PD, k)$;
   15 return $PS$;
```

At the coarse-grained phase, only *function entries* of the subject program are instrumented (Line 1). This sparsely instrumented program is then run against all the failing and a small portion of passing test cases to collect the coarse-grained profiles (Line 2). Each coarse-grained profile consists of a set of functions which are executed at least once during execution. Having these coarse-grained profiles, a coarse-grained analysis is performed to produce a list of functions with their respective *negative* and *positive supports* (i.e., the number of failing and passing coarse-grained profiles containing the function, respectively) (Line 3). This function list will guide the ensuing fine-grained phase.

The fine-grained phase has two passes, each of which performs fine-grained instrumentation followed by bug signature mining. The first pass is called "threshold boosting". It aims to set a threshold which will be used as a lower bound of the actual top-kth DS value of bug signatures mined by [11]. This is done by mining the top-kth DS value of signatures having only a small fraction of highly suspicious predicates instrumented (Lines 4-9). The second pass will efficiently produce the top-k bug signatures through safely pruning considerable predicates whose DS values are less than the threshold determined in the first pass (Lines 10-15).

Specifically, in the first pass, we first select a few predicates which are likely to be of high DS values (Line 4). The selection detail will be expounded in Section 3.2. Next, we perform the fine-grained instrumentation to instrument all these selected predicates (Line 5), and then run the instrumented program using all the failing and passing test cases to acquire fine-grained profiles (Line 6). The profiles are preprocessed to create the mining dataset (Line 7), which is fed into the bug signature miner to discover the top-k bug signatures (Line 8). Note that Lines 5-8 indicate the traditional procedure of predicated bug signature mining stated by [11], making it possible for modular plug-in of new debugging algorithm. After mining, we attain a threshold θ, which is the top-kth DS value of signatures mined during boosting (Line 9). We prove that this boosted threshold θ is a lower bound of the actual top-kth DS value of signatures mined in [11]. We will discuss this in Section 3.3. In the second pass, predicates with DS value lower than θ are pruned away, leaving behind a set of prospective predicates constituting the top-k bug signatures (Line 10), which will be discussed in Section 3.4. Only these prospective predicates are considered in the pruning pass. Lines 11-14 perform the bug signature mining as usual. Finally, the identical top-k bug signatures as mined by [11] are returned (Line 15). Note that since we have obtained the fine-grained profiles corresponding to the predicates in *boost* during the first pass, we only need to instrument other predicates in $prospect - boost$ in the second pass so as to further reduce execution time and storage space for profile collection (Line 11). However, we have to preprocess all the profiles (i.e., $PP+BP$) in order to perform *inter-procedural* signature mining (Line 13).

Identical test cases are used in coarse-grained phase (Line 2) and fine-grained phase (Lines 6 and 12). Except that in coarse-grained phase (Line 2), instead of all the passing test cases, only a subset of passing test cases[*] are used, thus further minimizing the cost of coarse-grained phase.

3.1 Instrumentation

A salient feature of the HI technique is to have multiple levels of instrumentation, where instrumentation at higher/coarser-grained level can help prune unnecessary instrumentation at lower/finer-grained level, resulting in big saving in performance cost. In this work, two levels of instrumentation are applied, namely coarse-grained (Line 1) and fine-grained (Lines 5 and 11).

At the coarse-grained phase, only the function entries across the program are instrumented (Line 1). Each function

[*]The number of passing test cases used is set to $\min\{\max\{0.1 \times |\mathcal{D}|, |\mathcal{D}^-|\}, |\mathcal{D}^+|\}$.

entry corresponds to one instrumentation site. After running the coarse-grained instrumented program over all the failing and a portion of passing test cases, we obtain a set of coarse-grained profiles each for one test case. Each profile records a set of functions which are executed during the run.

The instrumentation scheme discussed in Section 2.1 is used for the fine-grained instrumentation (Lines 5 and 11) where four types of predicates are considered. At each of the two passes, different parts of the program are instrumented, and the instrumented programs are executed using *all* the failing and passing test cases to generate two groups of fine-grained profiles, marked as failing and passing respectively. As mentioned in Section 2.1, each fine-grained profile is a set of predicates evaluated to true during the run. It corresponds to an itemset transaction in the fine-grained profiles database. Notice that our technique is also orthogonal to the instrumentation scheme. More types of predicates can be introduced without affecting our framework.

3.2 Predicate Selection for Boosting

Recall that the objective of the boosting pass is to generate a sufficiently high DS threshold of signatures for use in the pruning pass. Operationally, the threshold is generated by performing signature mining (*aka.*, MPS) on the subject with a small selected set of predicates being instrumented. As the whole mining process is involved, we wish to instrument as few predicates as possible so as to reduce overhead incurred and yet discover bug signatures with as high DS value as possible so as to prune more predicates away in the ensuing pruning pass.

We believe that the DS value computed during coarse-grained phase for each function, as an approximation to the DS values of the enclosing predicates, plays an important role in selecting predicates for boosting (i.e., the predicates instrumented during boosting pass). Specifically, we have the following hypothesis:

Hypothesis 1. *If the DS value of a function is high, then it is quite likely that the DS values of predicates within this function are high as well.*

In other words, there is a high correlation between DS values of a function and the predicates within. We test this hypothesis empirically by measuring the correlation coefficient (*i.e.*, Pearson's r [15, 16]) between the DS value of a function and the average DS value of all predicates within that function using 102 faulty versions in 5 buggy programs. For each version, we compute a correlation coefficient. Table 1 shows the averaged correlation coefficient among all the versions in each subject, excluding the statistically insignificant ones with p-value [15, 16] bigger than 0.05. The results indicate a strong positive correlation between DS values of a function and their predicates.

Table 1: Correlation Coefficient

Subject	replace	space	grep	sed	gzip	Overall
CC	0.69	0.71	0.63	0.46	0.71	0.64

Based on the above hypothesis, we elect to choose the predicates within functions of high DS values as the predicates for boosting. These predicates will be instrumented in the boosting pass for boosting a threshold. Specifically, we rank the functions in descending DS values, and select the predicates within the top few functions, until the total number of predicates selected reaches a pre-determined percentage γ of the total number of predicates in the entire program (Line 4). In our experiments, to guarantee low overhead, we set γ to be 5%.

3.3 Safeness of Threshold Boosting

We obtain a threshold θ, i.e., the top-kth DS value of signatures mined during boosting pass (Line 9). In the ensuing pruning pass, we will prune away those predicates whose DS values are less than θ, preventing them from being instrumented and thus saving the instrumentation and mining effort. To guarantee the safety of mining results (i.e., the actual top-k bug signatures mined by the original predicated signature mining [11] having all the predicates in the program instrumented, continue to appear in the results of pruning pass), we need to ensure that this boosted threshold is indeed a lower bound of the actual top-kth DS value (as stated by Theorem 3.2), such that no real top-k signatures will be erroneously missed. Before the proof, we first introduce a definition and a theorem, which will be used in pruning Theorem 3.2.

Definition 3.1 (**Projected Database**). *Given a set of distinct items $\mathcal{I}$, a subset $\mathcal{I}' \subseteq \mathcal{I}$, a set of class labels $\mathcal{C}$, and a class-labeled itemset databases $\mathcal{D}$ constituting n transactions, i.e., $\mathcal{D} = \{(T_1, c_1), \ldots, (T_n, c_n)\}$ where $\forall i \in [1, n], T_i \subseteq \mathcal{I} \wedge c_i \in \mathcal{C}$, a same-sized itemset database $\mathcal{D}' = \{(T'_1, c'_1), \ldots, (T'_n, c'_n)\}$ is said to be the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$ if and only if the following condition holds:*

$$\forall i \in [1, n], c'_i = c_i \wedge T'_i = T_i \cap \mathcal{I}'$$

Theorem 3.1 (**Pattern Preservation**). *Given a set of items $\mathcal{I}'$, two class-labeled itemset databases $\mathcal{D}$ and $\mathcal{D}'$ such that $\mathcal{D}'$ is the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$, for an itemset pattern $P \subseteq \mathcal{I}'$, the following holds:*[†]

$$DS(sup^+(P, \mathcal{D}'), sup^-(P, \mathcal{D}')) = DS(sup^+(P, \mathcal{D}), sup^-(P, \mathcal{D}))$$

Theorem 3.2 (**Lower Bound**). *Let θ be the boosted threshold, i.e., the top-kth DS value of signatures mined by boosting pass, ds_k be the top-kth DS value of signatures mined by the original predicated signature mining with all the predicates in the program instrumented, then we can derive that θ is a lower bound of ds_k, formally $\theta \leq ds_k$.*

Proof. Consider the original predicated signature mining, first instruments all the predicates in the entire program, then runs all the failing and passing test cases to collect profiles which are then constructed to a class-labeled itemset database for bug signature mining. As mentioned in Section 2.1, let $\mathcal{I}$ denote the set of all the instrumented predicates by the original signature mining, $\mathcal{D}$ denotes the class-labeled itemset database thus constructed. Accordingly, let $\mathcal{I}'$ correspond to the set of predicates instrumented at the boosting pass which is a subset of $\mathcal{I}$. $\mathcal{D}'$ is the itemset database derived during boosting. We can derive that $\mathcal{D}'$ is the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$. Since for each executed test case, we will attain the same labeled profile in both databases (i.e., let n be the number of test cases executed, then $\forall i \in [1, n], c'_i = c_i$). Meanwhile, for those predicates instrumented during boosting (i.e., $e \in \mathcal{I}'$), they are evaluated to true if

[†]The proof is provided by Appendix A.

and only if they are also evaluated to true in the original mining for the same test case. Therefore, these predicates will identically appear in both transactions $T_i \in \mathcal{D}$ and $T'_i \in \mathcal{D}'$, i.e, $\forall i \in [1,n], T'_i = T_i \cap \mathcal{I}'$.

Since $\mathcal{D}'$ is the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$, based on Theorem 3.1, we can conclude that for any itemset pattern mined in boosting pass, its DS value is identical to that computed in the original mining.

As mentioned before, a signature is a pattern consisting of itemsets with the same DS values. Therefore, given any signature mined in boosting, we will discover the same signature with the same DS value in the original mining. As a consequence, the actual top-kth DS value of signatures mined in the original signature mining ds_k will be at least the boosted threshold θ, i.e., the top-kth DS value of signatures mined during boosting pass, i.e., $\theta \leq ds_k$. $\square$

3.4 Predicate Pruning

In the pruning pass, we aim to discover the actual top-k bug signatures. We leverage the results from the coarse-grained phase (which is a list of functions associated with their respective negative supports) and the boosting pass (which is the boosted threshold θ) to safely prune away predicates whose DS value is less than θ, leaving behind a list of prospective predicates (Line 10). This is done by means of a safe pruning condition. Conceptually, the predicates within functions whose negative supports fall below a number n_0 (computed from θ) can be safely exempted from fine-grained instrumentation, since they cannot contribute a bug signature whose DS value is at least θ. Specifically, this safe pruning condition can be formalized as follows:

Theorem 3.3 (**Necessary Condition**). *Let e_i denote a predicate, m_i be the function in which e_i is located, $n(m)$ be the negative support of function m. Given a predicate itemset database $\mathcal{D}$, a bug signature P comprising a set of predicate itemsets all having the same negative and positive supports, for ease of presentation, let $P = \{\{e_1, e_2, ..., e_k\}\}$ and a threshold θ, the following implication holds where $n_0 = \arg\min_n\{IG(0,n) \geq \theta\}$:*

$$DS(sup^+(P,\mathcal{D}), sup^-(P,\mathcal{D})) \geq \theta \implies \forall i \in [1,k], n(m_i) \geq n_0$$

Proof. Let $p = sup^+(P,\mathcal{D})$ and $n = sup^-(P,\mathcal{D})$, we have the following deduction.

$$DS(p,n) \geq \theta \quad (3)$$
$$\implies IG(0,n) \geq \theta \quad (4)$$
$$\implies n \geq n_0 \text{ where } n_0 = \arg\min_n\{IG(0,n) \geq \theta\} \quad (5)$$
$$\implies \forall i \in [1,k], sup^-(\{e_i\},\mathcal{D}) \geq n_0 \quad (6)$$
$$\implies \forall i \in [1,k], n(m_i) \geq n_0 \quad (7)$$

Inequality (4) holds because $IG(0,n)$ is an upper bound of $DS(p,n)$ as proved in Theorem B.1 of Appendix B. Since $\frac{\partial IG(0,n)}{\partial n} = \frac{1}{|\mathcal{D}|}\log_2\frac{|\mathcal{D}|-n}{|\mathcal{D}^-|-n} > 0$ always holds, $IG(0,n)$ is monotonically increasing with respect to n $(0 \leq n \leq |\mathcal{D}^-|)$. Inequality (5) is thus derived. If $sup^-(P,\mathcal{D})$ is no less than n_0, then for all the predicates $e_i \in P$, their negative supports are no less than n_0 (6). Note that if e_i is evaluated to true during one failing run, then m_i must be executed during the same run. That is, given the same failing test cases, $n(m_i) \geq sup^-(\{e_i\},\mathcal{D})$ holds. Therefore, (7) holds. As a result, we have proved Theorem 3.3. $\square$

Based on the above necessary condition, we prune away all the predicates within the functions whose negative supports are less than n_0. Predicates that are not pruned away are called *prospective predicates*. As mentioned earlier, these prospective predicates will be instrumented and mined in the pruning pass.

From Theorem 3.3, we know that any bug signatures with DS value no less than θ, have to have their constituent predicates coming from these prospective predicates. Moreover, we have proved that this boosted threshold θ is a lower bound of the actual top-kth DS value of signatures in Section 3.3. Thus, mining done at the pruning pass can discover all the actual top-k bug signatures.

4. EMPIRICAL EVALUATION

We have conducted an empirical evaluation of our approach using 102 faulty versions in 5 buggy programs on an Intel Core 2 Quad 3.0GHz PC with 16GB main memory running 64-bit Fedora 19. Table 2 lists the subjects used, number of faulty versions, lines of code, number of functions, number of instrumentation predicates, and the size of test suite used. Note that in our experiments, all the results displayed for each subject are the average values computed across all the faulty versions in that subject. Moreover, the number of top discriminative signatures to mine k is set to 1, and γ which is the percentage of predicates instrumented during the boosting pass is set to 5%.

Table 2: Characteristics of subject programs

Subject	Versions	LoC	Functions	Predicates	Tests
replace	31	564	21	22,412	5,542
space	34	6,199	131	461,566	13,585
grep	12	10,068	121	1,418,835	809
sed	16	14,427	163	2,377,612	363
gzip	9	5,680	90	3,741,611	213

The original predicated signature mining [11] is composed of four main steps, namely instrumentation, profile collection, preprocessing and mining. Since each instrumentation is only performed once and then the instrumented program can be run forever, its cost is not significant compared with the other steps (i.e., profile collection, preprocessing and mining). In the following, we mainly discuss the performance improvement during the other three steps. Note that in our approach these steps are performed in each of the two passes of the fine-grained phase. Nevertheless, in comparing these two approaches, the experiments show that our approach, combining both coarse-grained and fine-grained phases, is able to greatly save the execution time and disk storage space for profile collection as well as the time and memory cost during the preprocessing and mining steps. Specifically, we measure the improvement of our approach in reducing execution time and storage space during profile collection in Section 4.1. In Section 4.2, we compare with [11] in terms of time and memory consumption for preprocessing and signature mining. We abbreviate the original predicated bug signature mining [11] as MPS (Mining Predicated Bug Signatures), whereas our approach as HIMPS (MPS via HI).

4.1 Profile Collection

In this step, we run the failing and passing test cases to collect profiles. This corresponds to Lines 2, 6 and 12 in Algorithm 1. Here we consider two aspects of performance cost, namely the execution time for running the instrumented programs and the disk storage space used for profiles.

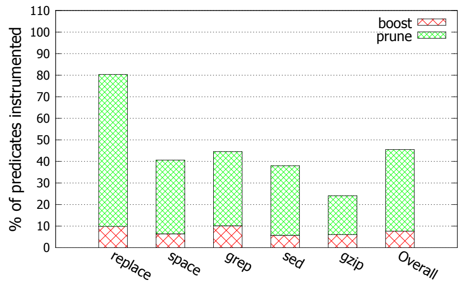


Figure 2: Percentage of predicates instrumented

As both performance costs are to some extent dependent on the number of predicates instrumented, we first present the percentage of predicates which are instrumented in our approach. Figure 2 depicts the percentage of predicates instrumented during boosting and pruning in HIMPS. As can be seen, compared with MPS where all the predicates (100%) in the whole program are instrumented, HIMPS manages to prune away considerable predicates. It only needs to instrument less than 45% of predicates for all subjects except for replace. Overall, more than half of the predicates are exempted from instrumentation compared with MPS. Note that HIMPS performs quite well especially on large programs. For gzip, even more than 75% of predicates are safely pruned away. For replace, it is of small size and most of the functions are executed during each run. HIMPS can hardly prune away predicates based on the coarse-grained information (i.e., negative and positive supports). As a consequence, it can only reduce 20% of predicates instrumented. We believe that the larger the program, the higher percentage of predicates our approach can prune away.

We have discussed the effectiveness of our approach in pruning unnecessary predicates in Figure 2. This provides an empirical evidence that HIMPS incurs relatively less time in executing instrumented programs and also utilizes less disk storage space for profiles than MPS. We further validate this hypothesis by directly measuring the execution time and the storage space for profiles.

Table 3: Execution time (in seconds) for profile collection

	MPS	HIMPS				Ratio
Subject	original	coarse	boost	prune	total	total/original
replace	12,332	841	6,829	10,108	17,777	144.16%
space	293,760	11,765	29,219	124,670	165,655	56.39%
grep	148,803	934	8,638	18,557	28,129	18.90%
sed	68,474	549	3,690	38,164	42,403	61.93%
gzip	663,789	1,945	112,151	57,980	172,076	25.92%
Overall	237,431	3,207	32,105	49,896	85,208	61.46%

Table 3 illustrates the execution time spent for running the instrumented program to collect profiles. We compare HIMPS including the coarse-grained phase (Column *coarse*) and two fine-grained passes (Columns *boost* and *prune*) with MPS adopting full instrumentation. As mentioned earlier, the coarse-grained instrumentation is lightweight. Moreover, only a small portion of passing test cases are executed in the coarse-grained phase. That is why Column *coarse* is quite small. Column *total* indicates the total execution time, which is the sum of all three columns in HIMPS. As can be seen, for grep and gzip, HIMPS only takes less than 30% of the execution time that MPS takes. But for replace, our approach costs more time than MPS due to running the fine-grained instrumented program twice. On average, HIMPS can save near 40% of the original execution time. Note that we employ *sampler-cc* [17] as the instrumenter in this experiment. Moreover, in order to ensure credible and stable results, we run each instrumented program four times, ignore the first run and compute the average execution time ~~of the other three runs.~~

三步相加

Table 4: Disk storage space used (in KB) for profile collection

	MPS	HIMPS				Ratio
Subject	original	coarse	boost	prune	total	total/original
replace	125,883	116	13,686	89,132	102,935	81.77%
space	6,242,337	1,950	401,591	2,174,017	2,577,558	41.29%
grep	1,145,575	170	116,008	391,439	507,617	44.31%
sed	864,367	125	49,515	288,398	338,038	39.11%
gzip	821,421	29	50,932	146,060	197,020	23.99%
Overall	1,839,916	478	126,346	617,809	744,634	46.09%

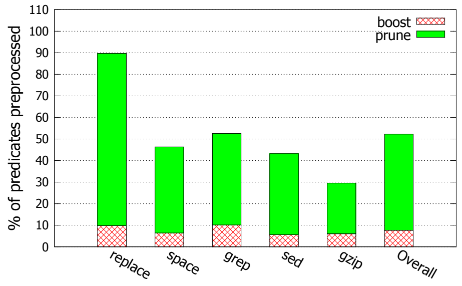
Table 4 presents the storage space used for profiles in kilobytes for HIMPS and MPS. Three groups of profiles were collected for HIMPS: the coarse-grained profiles (Column *coarse*), the fine-grained profiles during boosting (Column *boost*) and pruning (Column *prune*). It shows that HIMPS only requires less than 45% of the profile storage space required by MPS for all the subjects except for replace.

4.2 Preprocessing & Mining

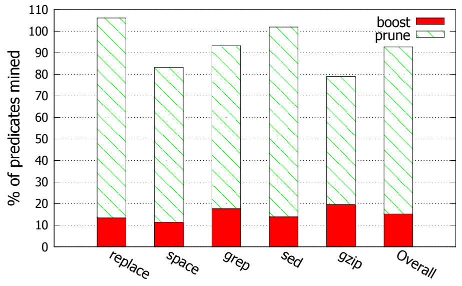
Having fine-grained profiles, we perform preprocessing to construct the mining dataset (Lines 7 and 13), and then mine bug signatures (Lines 8 and 14). Here, we compare HIMPS with MPS in terms of time cost and memory consumption for preprocessing and mining. Note that in our experiment, both HIMPS and MPS perform the same procedure as stated in Section 2.3 under the same setting.

We first demonstrate the percentage of predicates preprocessed and mined in HIMPS against that of MPS as the base (100%). Figures 3(a) and 3(b) plot the percentage during preprocessing and mining, respectively. As mentioned earlier, the profiles collected during boosting have to be preprocessed again in the pruning pass for the inter-procedural mining (Line 13 in Algorithm 1). That is why the percentage of predicates preprocessed (Figure 3(a)) is slightly bigger than that of instrumented (Figure 2). Nevertheless, around 50% of the total predicates are preprocessed during boosting and pruning together. Compared with 50% reduction in the number of predicates for preprocessing, overall only about 10% of predicates are reduced for mining shown as Figure 3(b). The underlying reason is that a certain number of predicates have been pruned through the filtering strategy during preprocessing discussed in Section 2.3, which weakens the pruning effectiveness of our technique. Again, HIMPS can prune away higher percentage of predicates for larger programs than for smaller ones. In the following, we directly provide the time cost and memory consumption for preprocessing and mining.

Tables 5 and 6 show the time cost in seconds and peak memory used in kilobytes for preprocessing and mining, respectively. Column *total* gets the sum of *boost* and *prune* for *Time* and the maximum for *Memory*. For preprocessing, HIMPS only takes at most around 40% of time that MPS



(a) preprocessing



(b) mining

Figure 3: Percentage of predicates analyzed

Table 5: Time (in seconds) and memory consumption (in KB) for preprocessing

	MPS		HIMPS						Ratio	
	original		boost		prune		total		total/original	
Subject	Time	Memory	Time	Memory	Time	Memory	Time	Memory	Time	Memory
replace	16.99	180,876	1.18	33,955	13.70	156,969	14.88	156,969	87.59%	86.78%
space	1,162.57	4,642,485	34.98	399,727	271.25	1,978,003	306.23	1,978,003	26.34%	42.61%
grep	128.63	895,591	9.33	164,821	41.64	466,192	50.97	466,192	39.62%	52.05%
sed	85.98	733,413	4.51	134,328	31.41	383,566	35.92	383,566	41.78%	52.30%
gzip	87.39	786,913	4.52	177,142	18.55	327,594	23.07	327,594	26.40%	41.63%
Overall	296.31	1,447,856	10.90	181,994	75.31	662,465	86.21	662,465	44.35%	55.07%

takes for all the subjects except for replace. The peak memory consumed is also smaller than 55% of that used by MPS. As for mining, HIMPS can also save more than 25% of time and 15% of peak memory consumption in general.

Table 7 demonstrates the total time cost and memory consumption for preprocessing and mining together. In addition, HIMPS also includes the time and memory used by the coarse-grained analysis, shown as Column *coarse*. We can see that the coarse-grained analysis is quite cheap compared with the fine-grained analysis (Column *boost* & *prune*). Overall, HIMPS can save more than 40% of total time and memory consumption compared with MPS for the whole analysis.

5. RELATED WORK

We provide an overview of literature on statistical debugging approaches in this section.

Statistical Bug Isolation. Statistical bug isolation approaches locate the root cause of failure by analyzing the discriminative behavior between passing and failing executions. The rationale is that program elements which are frequently executed in failing executions and rarely executed in passing executions are very likely to be faulty. Different types of elements are considered to represent the execution behavior in different approaches. In addition, various measures for assessing suspiciousness of elements are proposed. Tarantula [1, 2] as the first work, uses statement coverage information to represent the execution behavior and assesses the suspiciousness of each statement based on their proposed measure. Similar to Tarantula, Abreu et al. [5, 6] proposed Ochiai metric as the suspiciousness measure of program statements. Liblit et al. [3] collected runtime values of predicates and introduced *Importance* score to measure each predicate. SOBER [4] employs a different statistical model to evaluate predicates. Since the runtime predicate value captures finer-grained execution information than statement coverage, more precise results can be attained. However, it also suffers from heavy instrumentation cost and performance overhead. In [3], sparse random sampling [17] is adopted to address the overhead issue. As an extension of [3], Gore et al. [7] introduced elastic predicates such that statistical bug isolation is tailored to a specific class of software involving floating-point computations and continuous stochastic distributions. Furthermore, the causal inference has been recently applied to reduce the control and data flow dependence confounding bias in statement-level [18, 19] and predicate-level [20] statistical bug isolation.

Bug Signature Identification. As is well known, debugging is an integral process of localizing the bug, understanding and then fixing it. To assist in debugging, a great number of automatic debugging approaches have been proposed recently. Most of them focus only on the first phase which is termed as fault localization or bug isolation. These studies commonly try to isolate the root cause of the bug, which is usually a single buggy statement. However, in practice, it is difficult to understand the bug by examining that single statement in isolation. The *perfect bug understanding* does not hold [8]. To better support debugging, more information than sole buggy statement or root cause is required. The "context" where the bug occurs is likely to provide more useful clue for identifying, understanding and correcting bugs. Recently Hsu et al. [9] presented RAPID to identify bug signatures. They adopted sequence mining algorithm to discover longest sequences in a set of failing executions as the context. Jiang and Su [21] combined feature selection, clustering, and control flow analysis to identify faulty control flow paths that may cover bug locations. Cheng et al. [10] identified bug signatures using discriminative graph mining. They mined the discriminative control flow graph patterns from both passing and failing executions as bug signatures. Since only control flow transitions are considered in [10], bugs not causing any deviation in control flow transitions cannot be identified. To enhance the predictive power of bug signatures, Sun and Khoo [11] proposed *predicated bug signature mining*, where

Table 6: Time (in seconds) and memory consumption (in KB) for mining

	MPS		HIMPS						Ratio	
	original		boost		prune		total		total/original	
Subject	Time	Memory	Time	Memory	Time	Memory	Time	Memory	Time	Memory
replace	28.90	240,974	0.08	16,277	26.50	231,617	26.57	231,617	91.96%	96.12%
space	813.93	2,533,414	1.15	62,458	670.08	2,294,894	671.23	2,294,894	82.47%	90.59%
grep	300.15	321,836	1.82	23,908	166.89	245,349	168.72	245,349	56.21%	76.23%
sed	24.39	65,952	0.05	5,539	19.38	59,425	19.43	59,425	79.68%	90.10%
gzip	56.64	70,780	0.12	6,947	34.91	49,171	35.04	49,171	61.85%	69.47%
Overall	244.80	646,591	0.64	23,026	183.55	576,091	184.20	576,091	74.43%	84.50%

Table 7: Time (in seconds) and memory consumption (in KB) for preprocessing and mining together

	MPS		HIMPS						Ratio	
	original		coarse		boost & prune		total		total/original	
Subject	Time	Memory	Time	Memory	Time	Memory	Time	Memory	Time	Memory
replace	45.89	240,974	0.03	3,161	41.45	231,617	41.49	231,617	90.41%	96.12%
space	1,976.51	4,642,485	0.40	49,463	977.47	2,294,894	977.86	2,294,894	49.47%	49.43%
grep	428.78	895,591	0.73	47,906	219.69	466,192	220.41	466,192	51.40%	52.05%
sed	110.37	733,413	1.21	75,317	55.35	383,566	56.56	383,566	51.25%	52.30%
gzip	144.03	786,913	1.85	114,622	58.10	327,594	59.95	327,594	41.63%	41.63%
Overall	541.11	1,459,875	0.84	58,094	270.41	740,773	271.26	740,773	56.83%	58.31%

both data predicates and control flow information are utilized. They devised the discriminative itemset generator mining technique to discover the succinct predicated bug signatures.

Our HI technique aims to improve the efficiency of statistical debugging by performing selective instrumentation. It is independent of the particular statistical debugging approach (either statistical bug isolation or bug signature mining). Our technique can be applied to most of the debugging approaches [1, 2, 3, 5, 6, 7, 10, 11] discussed above to make them more efficient, only requiring deducing a respective necessary condition according to the specific fine-grained suspiciousness measure used.

6. CONCLUSION

In this paper, we introduce a novel *hierarchical instrumentation* technique to improve the efficiency of statistical debugging. This technique is based on the insight that *information collected and measured by instrumenting composite syntactic constructs (e.g., functions) can guide the selection of program elements (e.g., predicates) for subsequent fine-grained instrumentation*, resulting in significant saving in instrumentation effort, and substantial speed-up in the subsequent analysis process. We apply the HI technique to predicated bug signature mining and propose an approach called HIMPS. We provide evidence to support the insight mentioned above, and prove that HIMPS can safely discover top-k bug signatures. Our empirical study also concludes that in general HIMPS can achieve around 40% to 60% saving in disk storage space usage, time cost and peak memory consumption, and especially performs well on substantially large programs. This greatly improves the efficiency of predicated bug signature mining. Moving forward, we intend to formalize the HI technique and provide a more general and systematic approach for its application to statistical debugging.

Acknowledgments.

We thank Narcisa Milea, David Lo as well as his group members in SMU for their constructive discussions and questions during our meeting. We are also grateful to the anonymous reviewers for their valuable comments and advise on the paper. Our appreciation also goes to Ben Liblit at University of Wisconsin-Madison for making their predicate-based instrumentor available. This work is supported by an NUS Research Grant R-252-000-484-112.

7. REFERENCES

[1] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477.

[2] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282. 2005.

[3] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 15–26.

[4] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. In *Proceedings of the 2005 Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2005, pages 286–295, 2005.

[5] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.

[6] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, 2009.

[7] Ross Gore, Paul F. Reynolds, and David Kamensky. Statistical debugging with elastic predicates. In *Proceedings of the 2011 26th IEEE/ACM International*

Conference on Automated Software Engineering, ASE'11, pages 492–495.

[8] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209.

[9] Hwa-You Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 439–442. 2008.

[10] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 141–152, 2009.

[11] Chengnian Sun and Siau-Cheng Khoo. Mining succinct predicated bug signatures. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 576–586, 2013.

[12] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44. 2009.

[13] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[14] Jinyan Li, Haiquan Li, Limsoon Wong, Jian Pei, and Guozhu Dong. Minimum description length principle: Generators are preferable to closed patterns. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 409–414. 2006.

[15] Robert D. Mason, Douglas A. Lind, and William G. Marcha. *Statistics: An Introduction*. Duxbury Press, 5 Sub edition (1998), 1998.

[16] George Argyrous. *Statistics for Research: With a Guide to SPSS*. SAGE Publications Ltd; Third Edition edition (February 9, 2011), 2011.

[17] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 141–154.

[18] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 73–84, 2010.

[19] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *SIGSOFT FSE*, pages 146–156, 2011.

[20] Ross Gore and Paul F. Reynolds, Jr. Reducing confounding bias in predicate-level statistical debugging metrics. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 463–473.

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

APPENDIX

A. PROOF OF PATTERN PRESERVATION

The following provides the proof of Theorem 3.1.

PROOF. Since $\mathcal{D}'$ is the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$, we can derive that $\forall i \in [1,n], c_i' = c_i \wedge T_i' = T_i \cap \mathcal{I}'$ according to Definition 3.1.

Given an pattern $P \subseteq \mathcal{I}'$ and $\forall i \in [1,n], T_i' = T_i \cap \mathcal{I}'$, we have the following deduction:

$$\forall i \in [1,n], P \subseteq T_i' \tag{8}$$

$$\iff \forall i \in [1,n], P \subseteq T_i \cap \mathcal{I}' \tag{9}$$

$$\iff \forall i \in [1,n], P \subseteq T_i \wedge P \subseteq \mathcal{I}' \tag{10}$$

$$\iff \forall i \in [1,n], P \subseteq T_i \tag{11}$$

Thus we proved that $\forall i \in [1,n], P \subseteq T_i' \iff P \subseteq T_i$.

Further, recall that the positive support of P *wrt.* an itemset database $\mathcal{D}'$, $sup^+(P,\mathcal{D}') = |td^+(P,\mathcal{D}')|$ where $td^+(P,\mathcal{D}') = \{(T',c') \in \mathcal{D}' | P \subseteq T' \wedge c' = +)\}$. Since $\forall i \in [1,n], P \subseteq T_i' \iff P \subseteq T_i$ as proved above and given $\forall i \in [1,n], c_i' = c_i$, we can derive that $td^+(P,\mathcal{D}') = td^+(P,\mathcal{D})$ where $P \subseteq \mathcal{I}'$. Therefore, $sup^+(P,\mathcal{D}') = |td^+(P,\mathcal{D})| = sup^+(P,\mathcal{D})$. Similarly, we can get $sup^-(P,\mathcal{D}') = sup^-(P,\mathcal{D})$. According to Equation 2, $DS(sup^+(P,\mathcal{D}'), sup^-(P,\mathcal{D}'))$ will be equal to $DS(sup^+(P,\mathcal{D}), sup^-(P,\mathcal{D}))$. □

B. UPPER BOUND OF DS[‡]

THEOREM B.1 (UPPER BOUND OF DS). *Given an itemset database $\mathcal{D}$, and a pattern P, let $p = sup^+(P,\mathcal{D})$ and $n = sup^-(P,\mathcal{D})$, the discriminative significance of P, $DS(p,n)$ is upper bounded by $IG(0,n)$, shown as follows:*

$$DS(p,n) \le IG(0,n)$$

PROOF. According to the definition of *discriminative significance* (Equation 2), we prove it in the following two cases.

1. If $\frac{n}{|\mathcal{D}^-|} > \frac{p}{|\mathcal{D}^+|}$, according to Equation 2, $DS(p,n) = IG(p,n)$. The partial derivative of $IG(p,n)$ with respect to p:
$$\frac{\partial IG(p,n)}{\partial p} = \frac{1}{|\mathcal{D}|}\log_2\frac{p(|\mathcal{D}|-p-n)}{(p+n)(|\mathcal{D}^+|-p)}$$
Since $\frac{\partial IG(p,n)}{\partial p} < 0$ always holds where $\frac{n}{|\mathcal{D}^-|} > \frac{p}{|\mathcal{D}^+|}$, $IG(p,n)$ is monotonically decreasing with respect to p ($0 \le p \le |\mathcal{D}^+|$). Therefore, we can derive that $DS(p,n) \le IG(0,n)$.
2. If $\frac{n}{|\mathcal{D}^-|} \le \frac{p}{|\mathcal{D}^+|}$, according to Definition 2, $DS(p,n) = 0$. The partial derivative of $IG(p,n)$ with respect to n:
$$\frac{\partial IG(p,n)}{\partial n} = \frac{1}{|\mathcal{D}|}\log_2\frac{n(|\mathcal{D}|-p-n)}{(p+n)(|\mathcal{D}^-|-n)}$$
Since $\frac{\partial IG(0,n)}{\partial n} = \frac{1}{|\mathcal{D}|}\log_2\frac{|\mathcal{D}|-n}{|\mathcal{D}^-|-n} > 0$ always holds, $IG(0,n)$ is monotonically increasing with respect to n ($0 \le n \le |\mathcal{D}^-|$). Thus $IG(0,0) = 0 \le IG(0,n)$.

All in all, we proved that $DS(p,n) \le IG(0,n)$. □

[‡]The theorem name is same as one in [11]. But we introduce a totally different upper bound here for our use.