# Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs*

Chao Liu† Xifeng Yan†‡ Hwanjo Yu† Jiawei Han† Philip S. Yu‡

†Department of Computer Science
University of Illinois at Urbana-Champaign
{chaoliu, xyan, hwanjoyu, hanj}@cs.uiuc.edu
‡IBM T. J. Watson Research Center
psyu@us.ibm.com

**Abstract**

Analyzing the executions of a buggy software program is essentially a data mining process. Although many interesting methods have been developed to trace crashing bugs (such as memory violation and core dumps), it is still difficult to analyze noncrashing bugs (such as logical errors). In this paper, we develop a novel method to classify the structured traces of program executions using *software behavior graphs*. By analyzing the correct and incorrect executions, we have made good progress at the isolation of program regions that may lead to the faulty executions. The classification framework is built on an integration of closed graph mining and SVM classification. More interestingly, suspicious regions are identified through the capture of the classification accuracy change, which is measured incrementally during program execution. Our performance study and case-based experiments show that our approach is both effective and efficient.

## 1 Introduction

Software reliability is a top concern in modern industry. Software bugs cost the U.S. economy an estimated 59.5 billion dollars annually, or approximately 0.6% of the GDP, according to a report from the National Institute of Standards and Technology (NIST). As software becomes increasingly bulky in size, sophisticated in complexity, and originated by integration of multiple components, it is an increasingly challenging task to ensure software robustness and reliability.

As well-known in software engineering, better understanding of program behavior can be invaluable to build reliable systems. Extensive research has been conducted on software reliability, ranging from static source code checking [3, 6] to dynamic program verification [5, 18]; and from low-level program execution profiling [9, 7] to high-level behavior analysis [5, 20]. Related achievements have motivated practices in abnormality detection [9, 25] and computer-aided debugging [26, 18, 2].

From a knowledge discovery point of view, the analysis of executions of a buggy program is essentially a data mining process—tracing the data generated during program executions may disclose important patterns and outliers that may help the discovery of software bugs. Thus, we believe that recently developed data mining technology can improve software reliability. In this paper, we investigate the application of data mining methods to program bug analysis. By treating program executions as software behavior graphs, a new method is developed to integrate closed graph mining and SVM classification for the isolation of suspicious regions of noncrashing bugs.

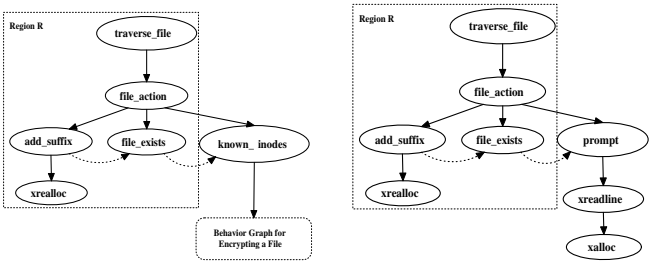
In program analysis, software bugs can be classified into two categories: *crashing bugs* and *noncrashing bugs*. The former refers to the bugs that crash the program execution, such as core dumps or segmentation faults. One can trace back the function call stack from the crashing point for debugging. The latter refers to the bugs that do not incur crashes, such as logic bugs, which are difficult to locate since no crashing point, hence no backtrace, is available.

In this study, we develop a novel classification method for backtracing noncrashing bugs. Our methodology can be outlined as follows.

First, we summarize each execution of a program as a concise but informative *behavior graph*. Fig. 1 shows an example of behavior graphs, which is excerpted from two different runs of *ccrypt-1.2*, a utility program for encrypting and decrypting files. Behavior graphs summa-

*This work was supported in part by the U.S. National Science Foundation NSF ITR-03-25603, an IBM Faculty Award, and an IBM Summer Internship. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

rize program execution at function level with each node for one function. Solid arrows represent the calling relationship and dashed ones for transitions. As one can see, behavior graphs only preserve function-level sequential information and are thus compact. Despite of its succinctness, it does manifest the behavior abnormities corresponding to incorrect runs. For example, *ccrypt-1.2* has one bug that is triggered when a user corresponds to the prompt for overwriting an existing file with EOF, rather than as expected 'Y(es)' or 'N(o)'. As shown in Fig. 1, the correct and incorrect runs diverge at the transition edges emitted from function `file_exists`, which is a strong indicator for classification.




(a) one correct run (b) one incorrect run

Figure 1: Software Behavior Graphs

Second, based on the behavior graph representation of program runs, the classification of program runs can be formulated as a graph classification problem: *Given a set of behavior graphs that are labelled either positive or negative, can we train a classifier to identify unknown behavior graphs?*

In our study, we use support vector machine (SVM) [13] with linear kernel to do classification. Inspired by the better scalability of closed subgraphs over frequent ones and their stronger expressibility over raw edges as features, we explore the benefits by incorporating closed subgraphs as classification features, which, as shown, has higher classification accuracy as well as better scalability. Interestingly, we also explore the relationship between closed and frequent graph-based SVM classifiers, which sheds light on the inherent relationship between these two related methods.

Third, for effective classification, we develop a novel method to uncover the "backtrace" for noncrashing bugs. Recall that backtrace usually refers to the function call stack at the time a program crashes (i.e., core dump or segmentation fault), based on which debugging can be easy to start. Unfortunately, for noncrashing bugs, such backtrace is no longer available. To help locate such bugs, we attempt to uncover a virtual "backtrace" for noncrashing bugs, which is essentially a series of bug-relevant functions. We believe that *the functions, whose execution behavior promotes the classification accuracy of distinguishing incorrect runs from correct runs, are likely suspicious functions.* Taking Fig. 1 as an example, a classifier can be trained at the return of function `file_exist`, but its accuracy cannot be high because behavior graphs up to this point (i.e., the subgraph within region R) are almost identical for both incorrect and correct runs. However, if we train another classifier at the return of `file_action` (recall that `file_action` returns later than `file_exist`), the accuracy will be much higher since the training behavior graphs do include the traces that differentiate correct and incorrect runs.

In summary, we make the following contributions:

1. We investigate the application of recently developed data mining techniques to software robustness enhancement and show that data mining may help backtrace noncrashing bugs.

2. We have proposed *software behavior graph* as a concise but informative summary of program executions and developed an efficient mining algorithm, *CloseMine*, to uncover closed frequent subgraphs from behavior graphs, which has been proven effective at identifying failing runs. We further explored the connection between closed frequent graph based and frequent graph based SVM classifiers.

3. We developed a novel classification method to uncover the backtrace for noncrashing bugs, which, as shown through a detailed case study, can be effective in assistance to debugging.

The remainder of the paper is organized as follows. We first introduce preliminary concepts in Section 2. The classification framework is laid out in Section 3, within which both the mining algorithm design and the relationship between frequent graph-based and closed graph-based SVMs are examined. Section 4 describes how to uncover a backtrace based on behavior graphs. Experimental evaluations of classification quality and a case study are presented in Section 5. We discuss the related work in Section 6, and conclude our study in Section 7.

## 2 Preliminaries

A software execution can be summarized into a *behavior graph*, which consists of its *call graph* and *transition graph*. A *call graph* $G_c(\alpha)$ is a directed graph displaying the function calling relationship in a program run $\alpha$. The vertex set $V(G_c(\alpha))$ includes all the functions involved in $\alpha$. Edge $(v_i, v_j)$ belongs to $E(G_c(\alpha))$ if and only if function $i$ calls function $j$ in $\alpha$. *Transition graph* $G_t(\alpha)$ is also a directed graph, but exhibits the

transition relationships in $\alpha$. Edge $(v_i, v_j)$ belongs to $E(G_t(\alpha))$ if and only if function $j$ is called immediately after function $i$ returns. It is also required that functions $i$ and $j$ are called by the same caller function. The superposition of $G_c(\alpha)$ and $G_t(\alpha)$ forms the behavior graph $G(\alpha)$ of run $\alpha$. Fig. 2 shows three behavior graphs, where solid and dashed arrows represent call relation and transition relation respectively.

We use behavior graphs to model program executions. Call graphs represent the task-subtask relationship, while transition graphs record the sequential order of the subtasks. Behavior graph only preserves the first-order transition and is thus succinct compared with the entire execution sequences. This is necessary for a scalable mining and classification method.

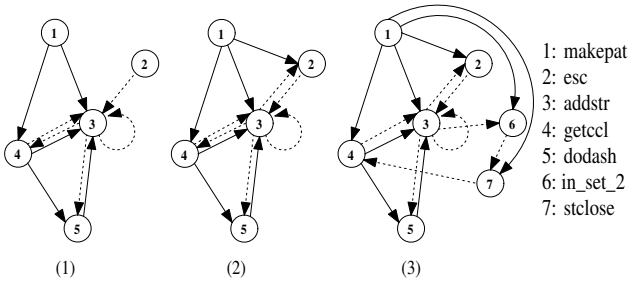



Figure 2: A Behavior Graph Dataset

Example 1. Fig. 2 shows behavior graph segments derived from three different runs of a program "*replace*", a regular expression matching and substitution utility software. Taking the run corresponding to the third graph for instance, `getccl`, `addstr`, `esc`, `in_set_2` and `stclose` are subtasks of function `makepat`. They work together to complete the task associated with `makepat`. As to transition, the dashed arrow from `getccl` to `addstr` means that `addstr` is called immediately after `getccl` returns. ∎

If a behavior graph $G$ is a *subgraph* of $G'$, then $G'$ is a *supergraph* of $G$, written $G \subseteq G'$. $G'$ is the *proper supergraph* of $G$ if $G \subset G'$. In the following discussion, we introduce the concepts of frequent and closed frequent graphs.

Definition 1. (Frequent (closed) graph) *Given a graph dataset $D$, support($g$) (or frequency($g$)) is the percentage (or number) of graphs in $D$, of which $g$ is a subgraph. A graph is frequent if its support is no less than a minimum support threshold, min_sup. A frequent graph is closed if there exists no supergraph that has the same support.* ∎

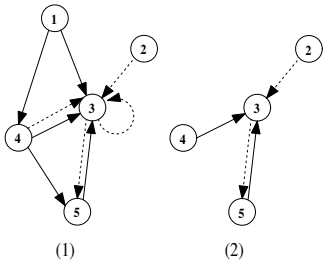



Figure 3: Frequent Graphs

Example 2. Fig. 3 depicts two of frequent subgraphs in the dataset shown in Fig. 1, assuming that min_sup is equal to 66.6%. In Fig. 3, the first graph is closed while the second is not since the latter is a subgraph of the former and both of them have the same support. ∎

## 3 The Classification Framework

Given a set of behavior graphs that are labelled either positive (for incorrect runs) or negative (for correct runs), we intend to train a classifier to identify new behavior graphs with unknown labels. The dynamics of classification accuracy will be analyzed to identify the backtrace of non-crashing bugs. In our study, we use support vector machine (SVM) [13] with linear kernel to do classification. The classification framework consists of three steps:

1. extract features from behavior graphs (training dataset),
2. learn an SVM classifier using these features, and
3. classify new behavior graphs.

In order to apply SVM in behavior graph classification, we represent graphs as vectors in a feature space. A naive representation is to treat *edges* as features and a graph as a *vector of edges*. The vector is $\{0, 1\}$ valued. If a graph has a specific edge, it has value "1" in the corresponding dimension, otherwise "0". Using this representation, the dot product of two feature vectors is the number of common edges that two graphs have

$$\mathbf{x}_i \cdot \mathbf{x}_j = |E(g_i) \cap E(g_j)|, \tag{3.1}$$

where $\mathbf{x}_i$ and $\mathbf{x}_j$ are the vector representation of graphs $g_i$ and $g_j$. For example, the dot product of the first two graphs in Fig. 2 is 10.

The similarity measure given in Eq. (3.1) is meaningful since it captures the relationship between two behavior graphs. As shown in our experiments, SVMs trained by the above measure work well in identifying some incorrect runs. Unfortunately, the hyperplane

learned in this way will be a linear combination of edges. Thus, it may not achieve good accuracy when a bug is characterized by multiple connected call and transition structures.

As shown in Fig. 2, the major portions of these graphs are very similar to each other although various incorrect runs may behave differently. In well-designed programs, functions usually exhibit strong modularity in source code and in dynamic executions. They are often grouped together to perform a specific task. Hence, the calls and transitions of these functions will be tightly related in the whole behavior graph. The buggy code may first disturb the local structure of a run and then have an effect on its global structure. This intuition inspires us to use recurrent local structures as features.

The classification process based on frequent graphs shares the same framework as the edge-based approach. Each frequent graph is treated as a separate feature in the feature vector. A behavior graph $G$ is first transformed into a feature vector whose $i$-th dimension is instantiated to 1 if $G$ contains the $i$-th frequent graph or 0 otherwise.

Unfortunately, due to the explosive number of frequent graphs in behavior graphs, it is often intractable to mine all of them. According to the Apriori property, all the subgraphs of a frequent graph must be frequent. A large frequent graph may generate a huge number of frequent subgraphs. When the number of frequent graphs increases, the performance at mining, training and classifying will drop dramatically. Thus, Deshpande et al. [4] propose a feature selection scheme to screen frequent graphs and Huan et al. [11] introduce the concept of coherent subgraphs to shrink the feature set. These approaches are successful in their problem domains. However, in our problem setting, they are not scalable. For example, in the "*replace*" program, if the minimum support is set at 40%, which is pretty high, there are still millions of frequent graphs. This renders the classification nearly impossible because it cannot even finish the feature extraction step.

As an alternative, closed frequent graph mining can complete in several orders of magnitude faster than frequent graph mining. Moreover, it commonly generates much less features for classification purpose. Taking the "replace" program as an example, among the millions of frequent graphs, only around 1,000 are closed frequent graphs. This makes the closed frequent graph-based classification more appealing than the frequent graph-based one. Furthermore, since closed frequent graphs is a lossless compression of frequent graphs, the classifier based on closed frequent graphs should have similar performance as the frequent graph based classifier. Our empirical study suggests that the former is better.

**3.1 Mining Closed Frequent Graphs.** The first step in our classification framework is to mine closed frequent graphs from a set of behavior graphs and then use them as features. Behavior graphs can be transformed to *labelled undirected pseudographs*. A *pseudograph* is a non-simple graph in which both loops and parallel edges are permitted. A *labelled graph* has labels associated with its vertices and edges. Since behavior graphs have distinct labels for each vertex, we can treat them as sets of 3-tuples $(v_i, v_j, elabel)$, where $i < j$. Edge label $elabel$ has four types: (i) uplink_call, (ii) downlink_call, (iii) uplink_transition, and (iv) downlink_transition, where "uplink" means that the edge direction is from $v_i$ to $v_j$ whereas "downlink" means the direction is from $v_j$ to $v_i$. In this way, each behavior graph is regarded as a set of distinct edges. Traditional closed graph mining algorithms, such as CloseGraph [24], do not take advantage of this property. In the following discussion, we develop a simpler graph mining algorithm that fits behavior graphs better.

We apply the pattern-growth methodology to mine closed frequent graphs[1]: Whenever a new frequent graph is uncovered, we extend this graph as much as possible until the maximum one is found. Let $g$ be a frequent subgraph with $n$ edges. Suppose $g$ is extended in a series of $g_1$, $g_2$, ..., $g_n$ ($g_1 = \varnothing$, $g_n = g$), where $g_i$ is a graph formed from $g_{i-1}$ by adding one new edge. If graphs $g_i, g_{i+1}, \ldots$, and $g_n$ have the same support, one could skip the search space between $g_i$ and $g_n$. That is, whenever $g_i$ is found, $g_i$ should be directly extended to $g_n$ through $g_{i+1}$ to $g_n$. Any other graph that is a supergraph of $g_i$ and a subgraph of $g_n$ should not be enumerated except $g_i, g_{i+1}, \ldots$, and $g_{n-1}$. We call it *search space skipping*. However, as illustrated in [24], CloseGraph has to miss some skipping in order to preserve the depth-first search order. The miss of search space skipping may cause problem when the closed frequent subgraphs are very large. Therefore, the naive search order [23] is adopted in our mining algorithm to skip the search space as much as possible.

Algorithm 1 (CloseMine) describes the pseudo code of our closed frequent graph mining algorithm. At each iteration of CloseMine, it first extends a newly discovered frequent graph with one more edge. Then CloseMine checks whether this graph has already been discovered (Line 1 in Algorithm 1). If not, it continues searching its supergraphs. CloseMine adopts an optimization

[1]Note that all the closed frequent graphs under examination are connected graphs.

**Algorithm 1** CloseMine($g$, $D$, minsup, $S$)

Input: A graph $g$, a graph dataset $D$, a minimum support threshold minsup.
Output: The closed frequent graph set $S$.

1: **if** $\exists\ g' \in S$ s.t. $g \subset g'$ and support($g$) = support($g'$) **then return**;
2: extend $g$ to $g'$ as long as $support(g) = support(g')$;
3: insert $g'$ to $S$;
4: scan $D$ once, find edge $e$ s.t. $g' \cup \{e\}$ is frequent;
5: **for each** frequent $g' \cup \{e\}$ **do**
6: CloseMine($g' \cup \{e\}$, D, minsup, $S$);
7: **return**;

(Line 2) that extends a frequent graph as much as possible until there is no supergraph having the same support.

**3.2 Relationship between Closed and Frequent Graph-based Classification.** In this section, we examine the relationship between the frequent graph-based and the closed frequent graph-based classification.

Since the whole set of frequent graphs can be reconstructed from closed frequent graphs, a potential question is whether frequent graph-based SVMs can be exactly constructed through a closed frequent graph-based training process? The answer is "yes". Actually, the concept discussed here can also be generalized to other kinds of frequent patterns like itemsets and sequences. Let us first examine how to build a mapping from frequent graphs to closed frequent graphs.

LEMMA 3.1. *Given a behavior graph $G$, there is one and only one closed behavior graph $G'$ such that $G \subseteq G'$ and $support(G) = support(G')$.*

**Proof.** Assume to the contrary that there is another closed graph $G''$ s.t. $G \subset G''$ and $support(G) = support(G'')$. Let $G^*$ be the graph formed by $G' \cup G''$. $G^*$ is a connected graph since $G'$ and $G''$ share a common subgraph $G$. Therefore, $G'' \subset G^*$ and $support(G'') = support(G^*)$, contradicting our assumption. ∎

Note that Lemma 3.1 only holds for graphs that have distinct label for each node. Fortunately, behavior graph has this property. Lemma 3.1 shows that there exists one function $f : \mathbb{F} \mapsto \mathbb{C}$, which maps any frequent graph in a frequent graph set $\mathbb{F}$ to one and only one closed graph in a closed frequent graph set $\mathbb{C}$. Thus, given a graph dataset $D$ and a pre-defined minimum support threshold $\delta$, the above mapping function can be obtained by mining closed frequent graphs from the dataset and constructing frequent graphs from closed frequent graphs.

In the *frequent or closed feature space*, a graph instance $G$ is represented by a feature vector whose $i$-th dimension is instantiated to 1 if $G$ contains the $i$-th feature (frequent graph or closed frequent graph) or 0 otherwise. Given a graph $G$, the vectors of $G$ in the frequent and closed feature space can be transformed with each other through the mapping function $f$ as described before. The number of frequent graphs that map to the same closed frequent graph $g$ is written as $c(g)$.

In the following discussion, we will show that an SVM trained in the frequent feature space for a training dataset can be constructed in the closed feature space. That is, we may solve the quadratic programming problem for a frequent graph-based SVM in the closed feature space.

Let $\mathbf{x}$ be the feature vector of a graph instance $G$ in the frequent feature space and $\mathbf{z}$ be the vector of $G$ in the closed feature space. Let $d$ be the number of dimensions in the closed feature space and $M$ be a diagonal matrix,

$$M = \begin{pmatrix} \sqrt{c(g_1)} & \dots & \dots & 0 \\ 0 & \sqrt{c(g_2)} & \dots & 0 \\ 0 & \dots & \dots & 0 \\ 0 & \dots & \dots & \sqrt{c(g_d)} \end{pmatrix}.$$

If we train a linear SVM in the frequent feature space, then $k(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$, which is equal to $(M\mathbf{z}_i) \cdot (M^T\mathbf{z}_j)$. Let $\mathbf{z}' = M\mathbf{z}$. Vector $\mathbf{z}'$ is in a new feature space $\mathbb{C}'$, which is formed by scaling the original closed feature space with $M$. Since $\mathbf{x}_i \cdot \mathbf{x}_j = \mathbf{z}'_i \cdot \mathbf{z}'_j$, the solution of the quadratic programming problem in this new space will be exactly the same as that of the quadratic programming problem in the frequent feature space. Thus, we may use closed frequent graphs as features with the scaling matrix $M$ to learn an equivalent SVM in the frequent feature space.

We further found that if two frequent graphs $g_i$ and $g_j$, $g_i \subset g_j$, are mapped to the same closed frequent graph, their weights in the optimal hyperplane are the same. That means SVMs cannot distinguish graphs $g_i$ and $g_j$ from the training set. In the closed frequent graph-based classification, we only treat graph $g_j$ as a feature (if $g_j$ is closed), while the frequent graph-based approach also counts $g_i$ as a feature. It is difficult to tell which method is better. However, our experiments indicate that the closed graph-based approach can achieve the similar or even better accuracy in comparison with the frequent graph-based approach.

## 4 Uncover "Backtrace" for Noncrashing Bugs

With the classification technique developed in Section 3, we here illustrate how to assist programmers in debugging noncrashing bugs.

Software bugs can be classified into two categories, according to their running behaviors. The first one is crashing bugs, which terminate the program execution abnormally with segmentation fault. For instance, illegal memory access and dereference to null pointers are two typical cases. Although crashing bugs happen quite often, they are not too difficult to tackle. At the crashing point, developers can obtain the backtrace, the snapshot of function call stack, based on which tracing back is straightforward. For example, in Fig. 1(b), the program crashes in `prompt`, then we have a function call stack, `traverse_file` → `file_action` → `prompt`. Programmers may carefully check the logic in these functions first. On the other hand, the other type is noncrashing bugs, which, as suggested through the name, do not incur program crashes. Noncrashing bugs are usually detected in software testing phase. Specifically, when a set of test suites are applied, some of outputs fail to match the expected. In general, fighting noncrashing bugs is harder than crashing ones. Few clues are available for programmers to debug noncrashing bugs.

Through comparison, we notice that this extra difficulty for noncrashing bugs partially comes from the absence of "backtrace"-like information. Suppose a "backtrace" is available for noncrashing buggy runs, which shows what functions are bug relevant, developers could be hinted to focus initial emphasis on those suspected functions. Therefore, we then aim at *identifying suspicious functions that are relevant to incorrect runs*. These functions may provide information to programmers in a way similar to "backtrace".

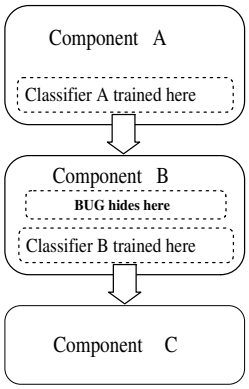



Figure 4: Classification Accuracy Boost

Our method is based on the analysis of the classification accuracy boost. Generally, the classification accuracy should not decrease while more and more trace data become available; especially, accuracy will improve once the execution data contain buggy behaviors. This is illustrated in Fig. 4. Suppose a program runs through components $A, B$ and $C$ in sequence and a noncrashing bug resides in component $B$ as shown. Classifier $f_A$ is trained at the end of execution of component $A$. As expected, its accuracy cannot be high since it knows few, if any, behaviors induced by the bug. In contrast, classifier $f_B$ that is trained after component $B$, is expected to have a much higher accuracy than $f_A$ because it does have behavior graphs induced by the bug in incorrect runs. Therefore, as long as $f_B$ has a classification accuracy boost in comparison with $f_A$, it is more likely that the bug is located in Component B than Component A. This inspires us to uncover "backtrace" for noncrashing bugs by detecting the accuracy change in a series of classifiers incrementally trained along the execution axis.

Specifically, for each function, $F_i$, two checkpoints $B^i_{in}$ and $B^i_{out}$ are placed at the entrance and the exit of $F_i$ respectively. At each checkpoint, a set of behavior graphs are collected, each of which corresponds to one test case running up to this checkpoint. Then using the classification technique developed in Section 3, a classifier can be trained at each checkpoint with accuracy (precision and recall) evaluated through cross-validation. In our experiments, we choose the highest precision as the accuracy measure while keeping recall no less than 90%. This guarantees that only few incorrect runs are missed and hence precision is a fair measure for comparison. In this way, each function is attached with a precision pair $[P^i_{in}, P^i_{out}]$. If there is a significant precision boost from $P^i_{in}$ to $P^i_{out}$, we would think function $F_i$ as bug-relevant. Its formal definition is given as follows.

Definition 2. (Bug-relevant) *Given a significance level of precision boost $\theta$ ($0 < \theta \leq 1$), a function $F_i$ is* bug-relevant *if $P^i_{out} - P^i_{in} \geq \theta$.* ∎

Consequently, bug-relevant function set (BRFS) refers to the set of functions that are bug-relevant with respect to a significance level $\theta$.

In general, BRFS is a smaller subset of all the functions, and hence it will be effective in helping programmers at debugging; otherwise, all functions are conceptually suspicious.

Furthermore, through experimental studies, we found that BRFS has several nice properties, which further enhance its applicability in debugging. For instance, it is easy to choose a proper cutoff $\theta$, distinguishing bug-relevant from "bug-irrelevant". In addition, due to the nested structure of function executions, BRFS is

| Version | Incorrect Runs | Correct Runs | Buggy Line # | Bug Description |
|---|---|---|---|---|
| 3 | 130 | 5412 | 493 | missing one condition testing in *if* testing |
| 4 | 143 | 5399 | 493 | misuse of variable |
| 5 | 271 | 5271 | 117 | misuse of < while <= is expected |
| 14 | 137 | 5405 | 369 | missing one condition testing in *if* testing |
| 26 | 198 | 5344 | 369 | misuse of *j* while *j+1* is expected |

Table 1: Summary of Buggy Versions

likely to line up in a form quite similar to "backtrace". However, since we are not yet very clear about the underlying model governing program executions, we refrain from presenting these properties formally. As an alternative, we examine a detailed case study in Section 5.4 together with reasonings about its soundness.

## 5 Experiments and Case Study

In this section, we evaluate the effectiveness and efficiency of closed frequent graph-based classification. A detailed case study is also given to illustrate its usage in uncovering backtrace of noncrashing bugs.

For classification evaluation, we designed three methods for comparison.

1. *edge*: Edges of a behavior graph are treated as features.
2. *frequent+*: In addition to *edge*, frequent graphs are treated as additional features. The symbol '+' means the classifier also uses edges as features.
3. *close+*: In addition to *edge*, closed frequent graphs are treated as additional features.

All of our experiments were carried out on a 3.2GHz Intel Pentium 4 PC with 1GB physical memory, running Redhat Linux 9.0. $SVM^{light}$ [13] was chosen in our implementation due to its good scalability.

**5.1 Subject Programs.** We took *Siemens Programs* as our testbed, which are widely used in software research [12, 21, 8, 10] because of its artificially instrumented but "realistic" enough software bugs. Readers interested in how Siemens researchers simulated realistic software bugs are referred to [12]. In our experiments, we chose *replace*, one of *Siemens Programs*, as our subject program. It performs regular expression matching and substitution. We chose it because the correctness of an execution is easy to label given the availability of a bug-free version.

*Replace* program in our study contains 32 versions in total, among which Version 0 is a bug-free version and other versions have one bug each. In this setting, Version 0 serves as the oracle in labelling whether a run is "correct". We conducted experiments on five buggy versions, which, in our point of view, nicely mimic the typical noncrashing bugs in reality. Table 1 shows the characteristics of these five buggy versions and their bug descriptions.

| Version | for incorrect runs | for correct runs |
|---|---|---|
| 3 | 15 | 549 |
| 4 | 22 | 547 |
| 5 | 74 | 538 |
| 14 | 39 | 604 |
| 26 | 50 | 543 |

Table 2: Number of Distinct Behavior Graphs

In order to objectively evaluate the effectiveness of classification, we remove duplicated behavior graphs within the set of correct and incorrect runs respectively. This is based on the consideration that two different but similar inputs may result in the same behavior graph. Table 2 lists the number of distinct graphs in the five versions.

**5.2 Effectiveness.** In our experiment, incorrect runs are labelled as positive samples and correct ones as negatives. As shown in Table 2, the numbers of positives and negatives are highly imbalanced, suggesting that we should evaluate the effectiveness through precision and recall, rather than pure accuracy.

*Recall* is defined as the fraction of the total number of incorrect runs that are classified right. *Precision* refers to the fraction of incorrect runs classified that are actually incorrect runs. Though it is highly desirable to achieve both high precision and recall, these two measures are usually contrary to each other. In practice, higher recall means low rate of missing incorrect runs while high precision means high hit rate and low rate of false alarms. In assistance to programmers' debugging, high precision with reasonably high recall means that the classification features are of high quality in discriminating incorrect runs from correct ones.

We perform five-fold cross validation and plot the result of each method in a recall-precision curve. Intuitively, a better method should have the recall-precision

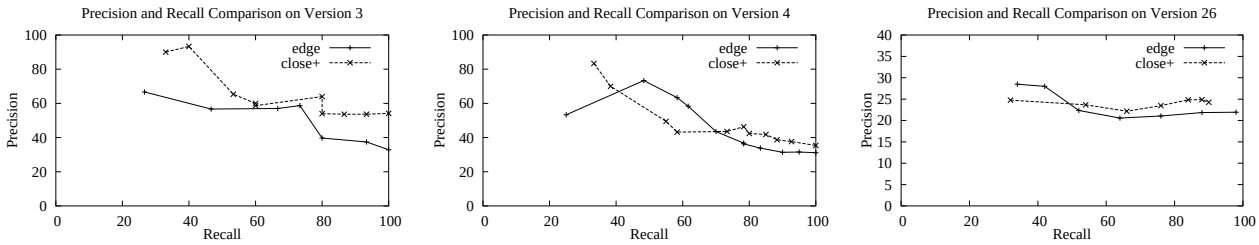



Figure 5: Precision and Recall: *close+* vs. *edge*

curve nearer to the upper right corner.

**5.2.1 Effectiveness in Detecting Failing Runs.** Fig. 5 shows the classification results of *edge* and *close+* on Versions 3, 4 and 26. The other two versions have the similar trend.

In classification of program runs, high recall is required due to the high cost of bugs [16, 1]. Thus we emphasize the precision when the recall is at a high level, e.g., 70% and higher. Version 3 in Fig. 5 has the best accuracy: with the 100% recall, the precision can be as high as 50%. It indicates that the classifier does not miss any real incorrect runs and at least one of two alarms is hit on average. Table 2 shows that the ratio between positives and negatives is about 1:37 (i.e., 15:549), which implies that random guessing according to this prior distribution would result in a precision around 2.7% (i.e., 15/(15+549)). The 20-times promotion of precision reaffirms our belief that behavior graphs are informative as to correctness of program executions. Similar conclusions can also be drawn on Version 4 and Version 26 depicted in Fig. 5. Generally, when the recall is as high as above 90%, our classifiers can still maintain a precision no lower than 25%. Considering the highly skewed distribution of positives and negatives in Table 2, we believe SVMs on behavior graphs perform well in the identification of incorrect runs.

Fig. 5 shows that *close+* generally outperforms *edge*, especially when a high recall is a must. This indicates that the addition of closed frequent graphs as features can leverage the classifier quality. In Versions 4 and 26, *edge* also achieves good performance. These are the cases where edges can be rather discriminative in revealing program correctness.

**5.2.2 Closed vs. Frequent Graph-Based Classification.** Next, we compare the classification accuracy between *frequent+* and *close+*. In Section 3, we show that frequent graph-based SVMs can be trained in the closed feature space. Therefore, we conjecture that

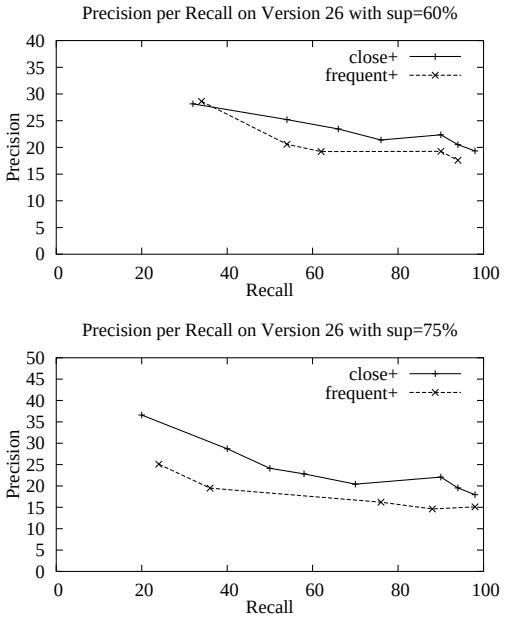



Figure 6: Precision and Recall: *close+* vs. *frequent+*

closed and frequent graph-based SVMs would probably have similar classification performance.

Fig. 6 presents the accuracy of *close+* and *frequent+* on Version 26, which also suggests a little bit better performance of *close+*. Note that the minimum support is set at 60% and 75% respectively, rather than 25% as used in Fig. 5. Under the 50% threshold, *frequent+* failed to complete the mining process.

**5.3 Scalability.** Figs. 7 and 8 compare *close+* and *frequent+* in terms of mining and training time. It indicates the better scalability of *close+* over *frequent+*. We only plotted the results from Version 3 for examination since others have the similar characteristics.

It becomes obvious that the computational cost of *frequent+* is exponential with regard to the minimum support threshold. For example, *frequent+* cannot finish in 10 hours when the support threshold is at 50%. On

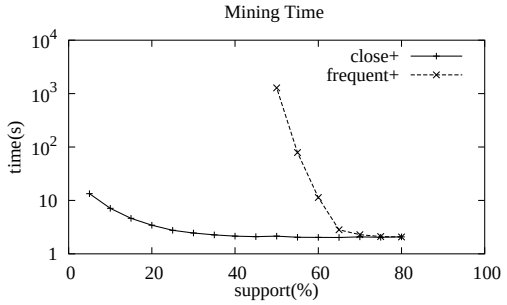



Figure 7: Mining Time w.r.t. Support

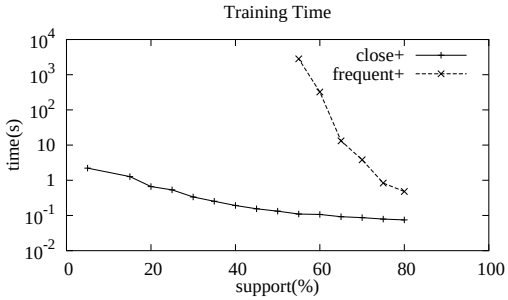



Figure 8: Training Time w.r.t. Support

the contrary, *close+* ran smoothly when the support was gradually lowered down. In practice, a reasonably low threshold is preferred since more patterns can be explored as potential features. When the support is at 5%, *close+* only takes around 15 seconds to learn a classifier (i.e. mining time + training time), which is surprisingly fast.

**5.4 Case Study.** In this subsection, we illustrate how to backtrace a noncrashing bug through a detailed case study. Section 5.4.1 describes what the bug is, followed by an examination of our approach in Section 5.4.2. We discuss its validity in Section 5.4.3.

**5.4.1 Case Description.** The buggy code we studied is shown in Program 1, which comes from Version 3 of the "*replace*" program. Within the if-statement at line 9, the subclause "`(lastm != m)`" is missed for some reason. This "miss of corner case" logic bug causes more than expected runs fall into the condition block between Lines 9 and 12, which in consequence induces incorrect outputs. In this buggy program, programmers may feel confused about where to start debugging since incorrect runs will finish smoothly. Usually they have to verify the code step by step, which is very time-consuming.

**Program 1** Buggy Code - Subline Function

```
1  void
2  subline(char *lin, char *pat, char *sub)
3  {
4    int i, lastm, m; 8
5    lastm = -1;
6    i = 0;
7    while ((lin[i] != ENDSTR)) {
8        m= amatch(lin, i, pat, 0);
9        if (m >= 0) /* && (lastm != m) BUG!!!*/{
10           putsub(lin, i, m, sub);
11           lastm = m;
12       }
13       if ((m == -1) || (m == i)){
14           fputc(lin[i],stdout);
15           i = i + 1;
16       } else
17           i = m;
18   }
19 }
```

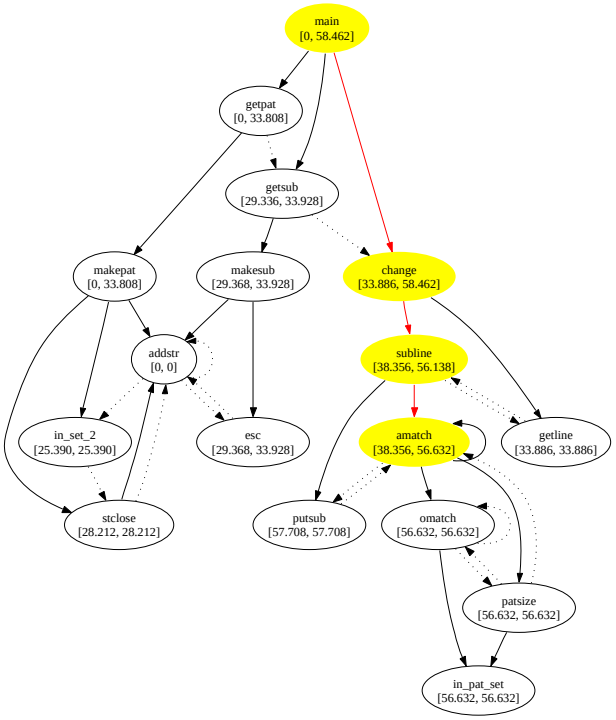



Figure 9: Entrance Precision and Exit Precision

**5.4.2 How It Works.** Fig. 9 shows the experimental results using our approach that helps narrow down the suspicious bug region. The classifiers are trained on behavior graphs from various program runs. Our classification method is applied at the entrance and the exit of each function. So each function has two precision values – entrance precision and exit precision.

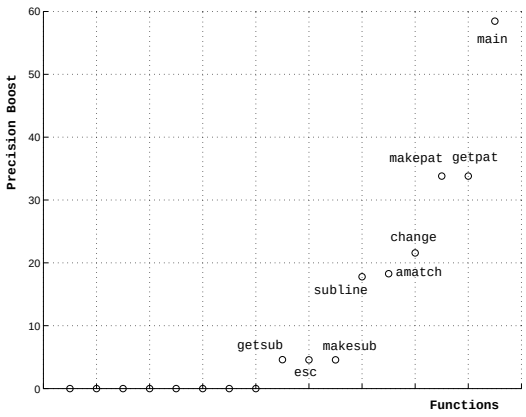



Figure 10: Precision Boost of Functions

Precisions depicted here are with recall at least 95%. We sort functions in increasing order of precision boosts in Fig. 10.

According to the method laid out in Section 4, the first task is to choose a proper significance level $\theta$ to identify bug-relevant functions. Seen from Fig. 10, eight functions induce no precision boost while another three only cause less than 5% precision increase. In contrast, the remaining six functions possess more than 17% boost. Therefore, it is easy to choose a safe cutoff in differentiating bug-relevant functions from irrelevant ones. The wide range of cutoffs clearly shows that bug relevance is an objective fact, rather than a subjective judgement. In addition, we believe this property should hold in general because functions that have nothing to do with the incorrect executions are less likely to cause significant precision boost. As a result, six out of the entire 17 functions are identified as bug-relevant. The result is summarized in Table 3.

| function name | $Precision_{in}$ | $Precision_{out}$ |
|---|---|---|
| `main` | 0 | 58.462 |
| `getpat` | 0 | 33.808 |
| `makepat` | 0 | 33.808 |
| `change` | 33.886 | 58.462 |
| `subline` | 38.356 | 56.318 |
| `amatch` | 38.356 | 56.632 |

Table 3: Bug-Relevant Functions with $\theta = 20\%$

Table 3 together with Fig. 10 exposes the following interesting results.

First, `main` function always has the highest precision and precision boost. This makes sense because $P_{out}^{\texttt{main}}$ measures the classifier that uses the information of the whole run, hence achieves the best precision. Meanwhile, $P_{in}^{\texttt{main}}$ is always 0% since no information is available in the entrance to the `main` function. Therefore, the `main` function always has the highest precision boost. Since `main` is the only entrance of a program, it is trivial to be regarded as bug-relevant.

Second, we can divide the six functions in Table 3 into two groups and rank them by their exit precision (i.e. $P_{out}^{i}$). Clearly, functions `main`, `change`, `subline` and `amatch` form a group with the highest exist precision, which actually reveals the backtrace to the buggy code.

Finally, bug-relevant functions tend to line up to form a backtrace. As shown in Fig. 9, the identified bug-relevant functions, namely `main`, `change`, `subline` and `amatch`, form the backtrace for this noncrashing bug. Again, we think this property should hold in general because the nested calling structure is typical in program executions. For instance, if function $A$ calls $B$ and $B$ is regarded as bug-relevant, $A$ would also be bug-relevant because $A$ exits later than $B$ and hence has more bug-relevant information.

In summary, through the above analysis we have uncovered the "backtrace" for this noncrashing bug. Taking this "backtrace" as hints, a programmer can start debugging in a similar way as facing the real backtrace. It is expected that a programmer could pay more attention on this backtrace rather than suspecting all the functions.

**5.4.3 Discussion on General Validity.** Although our method works reasonably well in the above case, we are not going to claim its general applicability. Due to the wide variety of software bugs, it is unlikely for a method to work well in all cases. In this study, we have been exerting great efforts to narrow down suspicious bug trace by using data mining techniques. The entire framework of exploiting classification dynamics to uncover "backtrace" makes sense by intuition and reasoning. Furthermore, our case study does capture a kind of common bugs, which may imply its applicability beyond this particular case.

We note that our method can only provide programmers with the "backtrace", a set of bug-relevant functions, which hopefully can assist programmers in a similar way as debugger-provided backtraces for crashing bugs. However, just as a real backtrace may not immediately lead to the discovery of the bug root for a crashing bug, neither does our method. Still a programmer has to scrutinize the source code and figure out a way to fix.

Computer-aided debugging is profound and hence hard to be solved thoroughly in one shot. To the best of our knowledge, it is less likely, if not impossible, to devise a fully-automated debugger, which detects and fixes

bugs without the involvement of human intelligence. We are looking forward to more debates and insights on this interesting and challenging problem.

## 6 Related Work

Previous related work falls into two fields: frequent pattern-based classification and software debugging.

**6.1 Frequent Pattern-Based Classification.** Statistical significance of frequent patterns motivates their applicability in classification problems, which is based on the belief that frequent patterns can embody significant and discriminative features. *Associative classification* [19, 17] tries to find a set of association rules based on frequent patterns, from which high quality rules are selected as meta-rules for classification. In contrast, we explore the potentials of *all* the patterns and use sophisticated learning algorithms, such as SVMs, to combine their discriminative power smoothly. In addition, *pattern-based classification* has been successfully applied to chemical and biological domains, such as classification of outer membrane proteins [22] and chemical compounds [4]. In this paper, we not only apply data mining techniques in software engineering, but also demonstrate the power by incorporating closed frequent patterns as features. As shown through experiments, our method has better scalability and meanwhile uplifts the classification accuracy. To the best of our knowledge, this is the first piece of work on using closed frequent patterns in classification and demonstrating their usage in software engineering.

**6.2 Software Bug Detection.** Software reliability is actively pursued in software engineering and computer system research from various angles. *Static analysis* [3, 6] aims at detecting program abnormities from the source code level without running the programs. *Dynamic analysis* [2, 5, 20, 18, 26], on the contrary, usually instruments subject programs to dump runtime information during their execution for further analysis. In addition, *model checking* [15] and *fault injection and analysis* [14] also work towards better software reliability through their own approaches.

Our work is in the category of dynamic program analysis, within which the following studies are the most related. Program invariants [7] are used to assist programmers in debugging [2, 18, 26]. Logistic regression is adopted in [18, 26] to single out discriminative invariants while Brun and Ernst use SVMs [2]. Researchers also explore the possibility of clustering incorrect runs based on software behaviors [5, 20]. We approach the software reliability problem through a classification method.

## 7 Conclusions

In this paper, we investigate the capability for computers to classify incorrect and correct executions based on observations of program behaviors. We develop a classification framework by summarizing program executions as behavior graphs. As demonstrated through experiments, the classification can be both effective and efficient. Moreover, we propose a novel method to exploit the classification accuracy boost and help programmers debug noncrashing buggy code, which otherwise may be elusive to handle. By examining software reliability from a data mining point of view, we make our initial efforts to explore how data mining techniques can contribute to software reliability, a hard but invaluable problem.

There are many issues that need to explore further. For example, it is not clear whether our method can be effective at tracing large software programs with the existence of multiple bugs in different program modules, how to further develop our method to make the trace deeper with finer granularity (such as a small set of program lines), and how to integrate this new approach with other existing software debugging methods. These are a set of issues for our future research.

## Acknowledgement

We would like to thank Professor Gregg Rothermel and his colleagues at University of Nebraska - Lincoln for providing us Subject Infrastructure Repository.

## References

[1] E. S. Agency. Arianne-5 flight 501 inquiry board report. In *http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf*.

[2] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. of the 26th Int. Conf. on Software Engineering (ICSE'04)*, 2004.

[3] D. Dhurjati, S. Kowshik, V. Adve and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proc. Languages Compilers and Tools for Embedded Systems*, 2003.

[4] M. Deshpande, M. Kuramochi, and G. Karypis. Frequent sub-structure-based approaches for classifying chemical compounds. In *Proc. of the third IEEE Int. Conf. on Data Mining (ICDM'03)*, 2003.

[5] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proc. of the 23rd Int. Conf. on Software engineering (ICSE'01)*, pages 339–348. IEEE Computer Society, 2001.

[6] N. Dor, M. Rodeh, and M. Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows

in c. In *Proceedings of the ACM SIGPLAN 2003 Int. Conf. on Programming language design and implementation (PLDI'03)*, pages 155–167. ACM Press, 2003.

[7] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[8] P. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT Int. symposium on Foundations of software engineering (FSE'98)*, pages 153–162. ACM Press, 1998.

[9] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th Int. Conf. on Software engineering (ICSE'02)*, pages 291–301. ACM Press, 2002.

[10] M. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90. ACM Press, 1998.

[11] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha. Mining spatial motifs from protein structure graphs. In *Proc. of the 8th Annual Int. Conf. on Research in Computational Molecular Biology (RECOMB'04)*, 2004.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th Int. Conf. on Software engineering (ICSE'94)*, pages 191–200. IEEE Computer Society Press, 1994.

[13] T. Joachims. Advances in kernel methods: Support vector learning. In *chapter Making large-Scale SVM Learning Practical*. MIT Press, 1999.

[14] G. Kanawati, N. Kanawati, and J. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44:248–260, 1995.

[15] S. Kumar and K. Li. Using model checking to debug device firmware. *SIGOPS Oper. Syst. Rev.*, 36(SI):61–74, 2002.

[16] N. Leveson and C. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[17] W. Li, J. Han, and J. Pei. CMAR: Accurate and efficient classification based on multiple class-association rules. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 369–376, San Jose, CA, 2001.

[18] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Proc. of the ACM SIGPLAN 2003 Int. Conf. on Programming Language Design and Implementation (PLDI'03)*, 2003.

[19] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proceedings of the fourth ACM SIGKDD Int. Conf. on Knowledge discovery and data mining (KDD'98)*, pages 27–31. ACM Press, 1998.

[20] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proc. of the 25th Int. Conf. on Software engineering (ICSE'03)*, pages 465–475. IEEE Computer Society, 2003.

[21] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transaction on Software Engineering*, 24(6):401–419, 1998.

[22] R. She, F. Chen, K. Wang, M. Ester, J. Gardy, and F. Brinkman. Frequent-subsequence-based prediction of outer membrane proteins. In *Proc. of the ninth ACM SIGKDD Int. Conf. on Knowledge discovery and data mining (KDD'03)*, pages 436–445. ACM Press, 2003.

[23] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, pages 721–724, 2002.

[24] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. 2003 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, pages 286 – 295, 2003.

[25] T. Zhang, X. Zhuang, S. Pande, and W. Lee. Hardware supported anomaly detection: down to the control flow level. In *Technical report, Center for Experimental Research in Computer System, GIT-CERCS-04-11, Georgia Institute of Technology*, 2004.

[26] A. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems 17 (NIPS'03)*, 2003.