Automatic Isolation of Cause-Effect Chains with Machine Learning

Lingxiao Jiang Zhendong Su

Department of Computer Science
University of California, Davis

{jiangl, su}@cs.ucdavis.edu

ABSTRACT

Accurate bug localization is important for automated debugging. One attractive approach is to apply statistical-based techniques on large number of evaluation profiles of program predicates to locate bug causes. Previous research has proposed a number of specialized techniques to isolate certain predicates as bug predictors. However, these techniques still require a programmer to manually examine significant fractions of code to locate bugs, and no existing statistical-based technique can discover cause-effect chains. In this paper, we adapt advanced machine learning techniques (such as support vector machines and random forests) and propose an effective algorithm for both isolating bug-related predicates and discovering cause-effect chains. Our technique combines feature selection (to select bug-related predicates) and clustering (to group similar predicates) in a novel way to precisely pinpoint bug causes. We have implemented our technique and validated it on various instrumented code including the Siemens test suite and `rhythmbox` (a music management application for `GNOME`). Compared to previous techniques, our algorithm not only discovers more bugs but also requires significantly less manual code inspection. For example, on the Siemens test suite, it discovers 38 bugs when programmers are expected to examine no more than 1% of the code, significantly better than 11, the best previously reported number (see Table 1, Section 5). Our algorithm also discovers previously unknown errors in `ryhthmbox` that were missed by previous techniques.

1. INTRODUCTION

Debugging is an important part of the software development process. Studies show that programmers spend majority of their time on testing and debugging. Debugging is traditionally a manual process and often done in the following way: A program exhibits unexpected behavior; the programmer examines the execution states of the program for causes of the problem. Such a manual task can be very tedious and challenging because the state space may be very large and may not even be completely available to the programmer, *e.g.*, in case of a failed user run.

It is thus desirable to automate the debugging process. Ideally, we would like to have an automated debugging system to locate and fix bugs automatically. This is perhaps unattainable, so more realistically, we would like automated debugging to filter out as much code unrelated to bugs as possible. The programmer then only needs to examine the remaining code to detect and fix the bug causes. This automatic filtering process is often refereed to as *bug localization*.

In recent years, much research has been devoted to automated debugging. For example, *statistical debugging* developed in the Cooperative Bug Isolation (CBI) project [19] is one simple, but promising approach. It has two components: (1) a predicate sampling mechanism to instrument deployed software and collect summarized profiles of program predicates from user executions and (2) several statistical metrics to select bug-related predicates [20, 21, 31]. The approach has been shown useful in bug localization. Within the CBI framework, Liu *et al.* recently propose a different statistical metric, SOBER, that achieves better results [22]. We delay the discussion of other related work to Section 8.

1.1 Our Approach and Contributions

So far, the suggested statistical debugging algorithms have been quite specialized. In addition, they cannot discover cause-effect chains and still require manual inspection of non-trivial fractions of the code. Our hypothesis is that advanced machine learning techniques can be used to develop effective bug localization algorithms, not only for isolating bug-related predicates but also discovering cause-effect chains. Bug localization is a natural machine learning problem. In this paper, we investigate the use of machine learning techniques for bug localization. We focus on two of the most successful algorithms: *support vector machines* (SVMs) and *random forests* (RFs).

We view a machine learning algorithm as both a *classification model* and a *clustering model*. As a classification model, the algorithm accepts a set of features and a set of data with class labels, and sets up the model for deciding the classes of all data points. For example, in our setting, a feature is an instrumentation predicate, a data point in the data set is the collected profile of a particular run, and a class label is a flag that indicates the outcome of a program run: success or failure. We call this a *vertical view* of the data because we compare different runs. From an established model, we can select those predicates that have greatest effects on class labels and thus are most relevant to the failures (see Section 2 and 3 for more details) to locate bugs.

We also view a machine learning algorithm as a clustering model. Given a data set, it clusters similar data points together. For our purpose, a data point here is different from that in the classification model: it is a collected profile of a particular predicate across all executions, both failed and successful runs. We call this a *horizon-*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Draft submitted to ICSE 2006, September 2005.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

tal view of the data because we compare different predicates. From the generated model, we can discover relationships among different predicates.

With classification alone, we can select bug-related predicates. However, we also need to discover how these predicates relate in terms of control flow in the program. Because classification may suggest only a few of the predicates, it is likely that from these predicates alone we may not be able to trace them in the control flow of the program (see Section 4 for concrete examples). We address this problem with a novel combination of classification and clustering: (1) we use classification to select bug-related predicates; (2) we use clustering to group related predicates; and (3) finally, we discover cause-effect chains of bugs by tracing clusters of bug-related predicates in the program's control-flow graph. This unique combination allows a programmer to focus his attention on the most relevant part of the program to the failure. To the best of our knowledge, no prior statistical-based techniques automatically discover cause-effect chains of bugs.

We have implemented our technique and validated it on code instrumented within the CBI framework [19], including `rhythmbox` and the Siemens test suite. We choose them for a direct comparison with earlier work on statistical debugging: `rhythmbox` was used by Liblit *et al.* within CBI [20, 21, 31] and the Siemens test suite was used in SOBER [22] and many other works on bug localization [4, 7, 25]. Under testing, our technique was effective; it precisely located many bug cause-effect chains and required examining only a small portion of the original code. For example, on the Siemens test suite, it discovers 38 bugs (out of 132) when programmers are expected to examine no more than 1% of the code, significantly better than 11 and 10, the best previously reported numbers (see Table 1, Section 5 for more details). Our algorithm also discovers previously unknown errors in `ryhthmbox` that were missed by previous techniques (Section 6).

1.2 Paper Outline

The rest of the paper is structured as follows. We first present the technical background for our approach (Section 2), including a brief introduction to the instrumentation framework CBI and necessary concepts on machine learning. Next, we introduce our machine learning-based bug localization technique (Section 3), followed by our experimental setup (Section 4) and detailed results (Section 5 and 6). Then we discuss the factors affecting our approach (Section 7). Finally, we survey related work (Section 8) and conclude (Section 9).

2. TECHNICAL BACKGROUND

2.1 Program Instrumentation Mechanism

Our technique relies on the CBI infrastructure to instrument programs and gather profiled information of program runs. Here is a brief introduction, and more details can be found in Liblit's work on statistical debugging [20,21]. The main observation is that information on program predicates is useful for understanding program (mis)behavior and that this information can be obtained from the large user base by distributing (lightly) instrumented code. For any particular program, CBI instruments the program to monitor three classes of predicates:

branches: To track the two predicates that indicate whether the true or false branches were taken.

returns: To track the sign of a function's return value.

scalar-pairs: To track the arithmetic relationship between two scalar variables or between a variable and a constant.

These different classes of predicates can indicate various kinds of bugs: branches are often used as error handlers when the program enters abnormal states; return values are often used to signal success or failure of functions; and scalar-pair relationship often relates to boundary issues that are responsible for many bugs.

These instrumented programs can be run in-house or by remote users. For each execution, a counter vector will be generated. Each element in the vector is the times that the corresponding predicate is true during the execution. A class label is also generated to indicate whether the execution fails or succeeds. Such information is then sent back to a central server for further analysis. In real deployment, in order to reduce runtime overhead of instrumented programs, not all predicates occurred in an execution are counted. CBI uses random sampling to decide whether a particular instrumentation should be executed or not [20]. Liblit *et al.* find that a sampling rate of $\frac{1}{100}$ keeps instrumentation overhead low, usually unmeasurable, while preserves enough information for analysis.

2.2 Machine Learning

We next introduce the main concepts of machine learning [24] that are relevant for our bug localization approach.

2.2.1 Definitions

For a machine learning problem, we are given a data set U: (1) Each data point $u \in U$ is a value vector $\langle v_1, v_2, \ldots, v_n \rangle$ for a set of pre-selected features $P = \{p_1, p_2, \ldots, p_n\}$; and (2) Each data point u has an associated class label C_u that indicates to which class u belongs. In our setting of bug localization, v_i denotes the number of times that the corresponding predicate p_i is observed to be true in an execution, and C_u indicates failure or success of the execution represented by u.

There are three common machine learning tasks:

classification: Given U and P, establish a model $M(u)$ to map each $u \in U$ to a class, maximizing classification accuracy, *i.e.*, maximizing the number of correct mappings on another data set from the same sample space, even if U contains noise data (whose values should be irrelevant to the final model or whose C_u is incorrect) or data with missing values (whose values are incomplete).

feature selection: Given U and P, select from P a subset of features $P_k = \{p_{s_1}, \ldots, p_{s_k}\}$ such that the classification model based on U_k and P_k is still accurate enough, where U_k projects U onto P_k, *i.e.*,

$$U_k = \{\langle v_{s_1}, \ldots, v_{s_k} \rangle \mid \langle v_1, v_2, \ldots, v_n \rangle \in U\}$$

If k is restricted to a constant, we usually select a P_k with maximal classification accuracy, which is refereed to as *k-feature selection*. In this case, P_k is the set of features whose values exhibit the most difference among different classes. Thus these features have the most effect on the classification model. In terms of debugging, the selected predicates are most bug-related.

clustering: Given U without class labels and P, divide U into subsets (*clusters*) such that data in each subset are similar w.r.t. certain distance measure [2]. This is often referred to as *unsupervised learning* [11] because each data point has no associated class label, where classification is referred to as *supervised learning* [24].

2.2.2 Machine Learning Algorithms

Although many machine learning algorithms exist, the following two classes of algorithms achieve the highest classification accuracy so far. In this paper, we focus on these two algorithms for bug localization:

Support Vector Machines: *Support vector machines* (SVMs) [5] are a family of machine learning algorithms that are gaining popularity, partly because of their balanced trade-off between performance and accuracy for learning problems. Generally speaking, a SVM chooses a model, such as a linear function $u * \omega$ or a radial basis function $e^{-\gamma * ||u-\omega||^2}$ where ω and γ are pending model factors, and iteratively adjusts the model factors to achieve maximal classification accuracy while minimizing certain inevitable errors introduced by pre-selected models and adjusting model factors.

Random Forests: *Random forests* (RFs) [3] are a collection of *decision trees* (DTs) [24] with each tree built on a random subset of U, independently sampled with the same distribution as U, and a random subset of P with a fixed size m. The number of trees and m are chosen by users. After all trees are constructed, the classification model will be the following: given a data point $u \in U$, pass u down each of the trees in the forest; each tree gives a classification decision (a *vote*) and the class with the most votes is selected for u.

When dealing with large data sets, RFs are generally more accurate and have better performance. In addition, RFs have built-in heuristics for handling missing values in the data set, and are thus helpful to analyze randomly sampled data produced in the CBI infrastructure. On the other hand, SVMs are generally more accurate when the size of data sets are small.

We also experimented with DTs initially and found that both SVMs and RFs are much more accurate than DTs for locating bugs.

3. BUG LOCALIZATION FRAMEWORK

In this section, we present our machine learning-based framework for bug localization, including isolating bug-related predicates and cause-effect chains.

3.1 Overview

Figure 1 shows the architecture of our framework. First, programs are instrumented (within the CBI infrastructure), and sampled data for executions are collected (Section 3.2). After the data has been preprocessed, it is fed to machine learning algorithms, and two machine learning tasks are performed: feature selection to identify bug-related predicates (Section 3.3) and clustering to group similar predicates (Section 3.4). We then perform a control-flow graph (CFG) analysis to correlate all predicates in the same clusters (Section 3.5) to form a set of possible cause-effect chains. The final outcome of the framework is a set of possible cause-effect chains that contain selected bug-related predicates for locating and fixing the bugs.

3.2 Data Collection

Instrumented by CBI, each program counts the times each instrumentation predicate is true in its execution. There are several different kinds of instrumentation schemes for collecting data (see Section 2.1). Such counter information is what we use for locating bugs. Although such information is not as accurate as provided by program states, it can utilize the large user base to gain enough statistical information on the program while reducing bandwidth requirements on users. As data is accumulated, the results can get more accurate. Furthermore, such *abstract* data, in fact, facilitates the clustering phase, because it may help flatten out certain non-

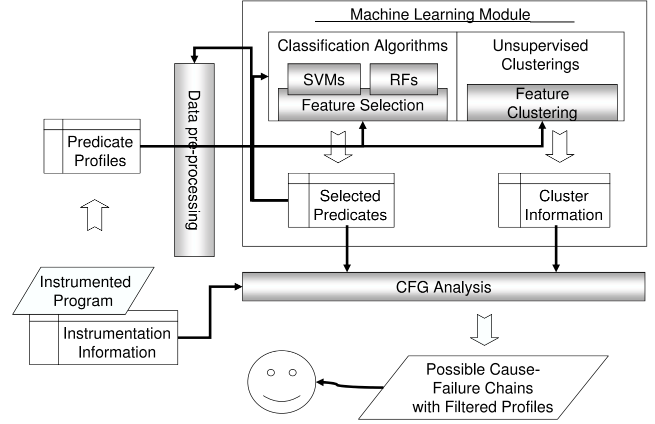

Figure 1: Architecture of Bug Localization Framework.

linear relations among *concrete* states to linear ones, which are usually much easier to be analyzed for similarity.

Collected data can be two kinds: one has class labels and the other has no class labels [1]. Although data without class labels can also be used for unsupervised machine learning [11], we use data with class labels here for establishing more accurate classification models.

We need to preprocess the gathered data before feeding it to machine learning algorithms. This is mainly to transform the data format, make up missing values in certain data points (we simply let them be zeros when necessary), project the data according to pre-selected predicates if required, and assign an identifier to each instrumentation predicate to determine the predicate's corresponding position in the source code.

3.3 Feature Selection

Different machine learning techniques usually have different criteria for selecting feature subsets. They are usually based on classification models. A naïve approach is to exhaustively search through all possible feature subsets of a particular size k (recall that k is a constant) for good candidates. A more practical approach, although may not be globally optimal, is to assign certain importance scores to every feature based on this particular feature's effects on the classification model. We can then select the top-k predicates for a k-feature selection.

For a SVM with a linear model $u*\omega$, it is straightforward to compute such scores. The classification model $M(u)$ here is usually a hyperplane with the normal vector ω. The i-th feature's effects on the model $eff_{p_i}(M)$ can be defined as ω_i^2 or simply $|\omega_i|$. For a SVM with a non-linear model, $eff_{p_i}(M)$ can be defined as the square of the partial derivative of $M(u)$ *w.r.t.*to the i-th feature [12], *i.e.*,

$$eff_{p_i}(M) \triangleq (\frac{\partial M(u)}{\partial u_i})^2$$

We experimented with SVMs based on both linear and radial basis functions. The results indicate that the classification accuracy of the respective models and selected predicate subsets is very close. We thus focus on using the more efficient linear models in our experiments

RFs internally assign importance scores to features during the

[1] It is possible when the exit status of a program cannot be determined.

construction of the forest [3]: (1) During the construction of a tree t, on average about one-third of all data points are left out of the sampled data for t and not used; these data are called *out-of-bag* (oob) data of t. In every tree t grown in the forest, pass all its oob data down and count the number n_1 of correct votes; and (2) Randomly permute the values of a feature m in the oob data, pass them down t again, and count the number n_2 of correct votes. The average of $(n_1 - n_2)$ over all trees in the forest is the *raw importance score* for m. Dividing the raw score by its standard error gives us a *z-score*, the metric for feature selection: a feature with a higher *z-score* means that feature-permutation on m exhibits greater impact on classification, and thus it is a more important feature.

Feature scores are usually not computed in SVMs because not all SVM models have a meaning way to assign scores; while for RFs, these scores are always computed, which makes RFs more attractive in this regard. In our experiments, we have implemented score assigning for linear SVMs, and use both SVMs and RFs to select bug-related predicates and help perform bug localization.

As an optional step, we can project the original data set onto the bug-related predicates (or those other predicates clustered with bug-related predicates), and then re-establish classification models to iteratively improve the accuracy of the models for bug localization. We also experimented with this step, and did not observe significant benefits. Thus, we chose not to use such iterative steps in our experiments.

3.4 Clustering

Much work has been done on discovering relationships among different executions (the vertical view) and the relations among program predicates and executions (predicate selection), but little has been done on discovering relationships among predicates across different executions (the horizontal view). Predicate selection does not explicitly take the relationships among predicates into account. However, program faults may be caused by or influence many predicates in an execution. Such correlations among predicates can provide additional information about bugs. For example, we can exploit the transitional information provided by such relationships to expose more bugs and discover cause-effect chains. Therefore, we want to cluster predicates related to the same bug(s) together.

In order to perform clustering, we need a metric for measuring the similarity among predicates. A natural metric can be based on the similarity of value distributions for different predicates, and such a similarity measure can be casted as testing statistical hypotheses [16], similar to the statistical metric used by SOBER [22]. If the value distributions of two predicates across all executions are similar, the two predicates are likely to be related. An alternative metric is the revised L_1 distance between two vectors. A vector here is comprised of all values of a predicate in all executions: $V_P = \langle R_{1P}, \ldots, R_{NP} \rangle$, where N is the total number of executions and R_{iP} is the value of predicate P in the ith execution. We define the revised L_1 distance between two vector V_{P_1} and V_{P_2} as the following:

$$D(V_{P_1}, V_{P_2}) \triangleq \frac{\Sigma_{i=1}^{N} |S_{iP_1} - S_{iP_2}|}{N} \quad (1)$$

where S_{iP_1} and S_{iP_2} are respectively R_{iP_1} and R_{iP_2} linearly scaled to $[0, 1]$, *e.g.*,

$$S_{iP_1} = \frac{R_{iP_1} - \min_j(R_{jP_1})}{\max_j(R_{jP_1}) - \min_j(R_{jP_1})} \quad (2)$$

During clustering, we set an ϵ parameter, such that the distance between each predicate and the mass center of its cluster (the average of all data points in the cluster) is less than $\frac{\epsilon}{2}$. This idea is similar to k-means clustering [2] and guarantees that the distance between any two predicates in the same cluster is less than ϵ. Thus, when two predicates are in the same cluster, we can say with high confidence that they are linearly related, and thus either both or neither of them are bug-related.

3.5 Discovering Cause-Effect Chains

After predicate selection and clustering, if a cluster contains any of the bug-related predicates, we consider it as a candidate for generating *cause-effect chains*: control flow paths that connect bug causes and bug-related predicates. Given a candidate and the positions of all the predicates in the cluster, we analyze the control flow graph (CFG) of the original program and find those paths that connect all the predicates. Such paths are usually not unique, thus we identify the *shortest* one as the cause-effect chain for manual inspection. It is possible that our control flow analysis cannot find a path connecting all the predicates; several chains may be reported for a cluster. Chains reported for all clusters are reported back to the programmer in the order of their lengths for inspection. Concrete examples can be found in Section 5.

As a heuristic for determining the cause-effect chains, we utilize the nature of instrumentation predicates for the `branches` scheme to prune control flow paths. Recall that the value of a predicate is the number of times that the predicate is true in an execution. We name a few heuristics that we use: (1) If most values of a predicate are zeros in failed runs, and most values of the negation of the predicate are non-zeros, it means bugs are more likely to be on the paths when the predicate is false; (2) If both the values of a predicate and its negation are zeros in failed runs, it means that the paths related to the predicate are not executed; and (3) If both are non-zeros, we can track either the true or the false case. We could have divided the cases finer, but in practice we find such heuristics could already help us choose shorter cause-effect chains.

Based on the results of predicate selection and clustering, such chains will most likely contain all bug-related predicates, and presenting them in the form of control flow paths can thus help programmers quickly locate and understand bugs.

One may be tempted to treat all selected predicates as a cluster and try to connect them and find cause-effect chains. However, such a cluster has less predicates and thus provides less hints as how the predicates are related. First, this would require more resource to track alternative control flow paths. Second, the chosen cause-effect chains would also be too approximate and may miss real bugs. Our experiments confirmed this observation (see Section 5.2). Thus, combining predicate selection and clustering is essential for efficiently discovering accurate cause-effect chains.

4. EXPERIMENTAL SETUP

In this section, we discuss details of our experimental setup. We use mainly two machine learning tools: (1) Experiments with SVMs are performed with a support vector machine classification tool—*LIBSVM* [6]—on a machine with a 2GHz Intel Xeon processor and 1GB RAM, running Linux kernel 2.6.11; and (2) Experiments with RFs are performed with an evaluation version of RandomForests [28] on a machine with a 2GHz Intel P4-M processor and 512MB RAM, running Microsoft Windows XP Professional. These implementations of SVMs and RFs are memory-based and cannot handle data set larger than the available memories on the respective machines. The evaluation version of RandomForests supports at most $8M$ data with 32768 predicates. We implemented the linear scaling and the clustering strategies discussed in Section 3.4 ourselves.

4.1 Data Collection

Previous work on bug localization [4, 7, 22, 25] often uses the Siemens test suite [14] or modified versions of the suite, *e.g.*, by Rothermel and Harrold [27], or by Renieris and Reiss [25]. The suite is a de facto benchmark for bug localization techniques.

CBI has also accumulated large sets of data for many programs [19], including the data set for `rhythmbox` (a music management application for GNOME), which has been used to discover some interesting bugs in `rhythmbox` [17, 18, 26]. In contrast to the Siemens test suite, `rhythmbox` makes heavy use of threads that makes control flow analysis more difficult. We choose to use them in our experiments for a direct comparison with previous work.

We use the HR variants of the Siemens suite [1]. It contains 132 faulty versions of seven programs. Each program has thousands of test cases and from zero to hundreds of failed runs for analysis. We use program outputs to decide whether a test case succeeds or fails. Some statistical data on the test suite can be found in Graves *et al.*'s study [10].

We instrumented the Siemens test suite with CBI (without sampling), and collected data from the executions of all its test cases. The data of `rhythmbox` was provided by Ben Liblit. Although it is for an old version (0.6.4), it is still meaningful for us to evaluate the effectiveness of our approach. In fact, we have discovered bugs that still exist in newer versions of `rhythmbox` and reported them to the developers.

4.2 Data Preprocessing

The whole data set of `rhythmbox` contains about $432,000$ instrumentation predicates and $32,000$ executions. The LIBSVM and RandomForests cannot handle this large size data set. Thus, we randomly divide the whole data set into several subsets for analysis. Our experiments show that using as few as thousands of executions that include around tens or hundreds of failed runs is sufficient to find interesting information about bug locations and causes. We also split each data set into three subsets according to the three different kinds of instrumentation predicates in order to exhibit the effectiveness of different instrumentation schemes on bug localization.

We use linear scaling here not only for clustering, but also for preventing predicates with larger ranges dominating others during predicate selection. Such scaling is also helpful for preventing computational overflows during the SVM and RF learning process.

4.3 Learning Parameters

First, we need to adjust the weight parameters for successful runs and failed runs because the data sets are *unbalanced* (*i.e.*, the number of successful runs is much larger than the number of failed runs). A general guideline for adjusting the weight parameters is to set the weights according to the ratio between the number of data from different classes. LIBSVM [6] and RandomForests [28] are both able to adjust the weights accordingly.

Second, random forests require the number of trees in a forest and the number of predicates used for constructing a tree as parameters. The principle for choosing such parameters is to decrease the classification error or to make it *stable* (*i.e.*, the classification error of a forest always converges when the number of trees increases). Based on the data size and the computing resources we have available, we choose 500 unless it is obvious that we have not reached the convergence. We use the square root of the total number of predicates as the number of predicates for constructing a tree, and halve or double it if classification errors are high or not stable.

Third, we decide to use 3-feature selection (*i.e.*, $k = 3$) for both SVMs and RFs. In addition, if a predicate's score is more than two times the next one's in the list of all predicates sorted by their scores, we can discard the rest of the list with high confidence; if more than three predicates have the top-3 scores, we then choose more predicates.

Finally, clustering requires ϵ, which may affect the final clusters and thus cause-effect chains. Based on our definition of distance (*cf.*Section 3.4), the distance of two un-scaled data values, $\epsilon \times \langle value\ ranges \rangle$, should be intuitively small if they are clustered together. In the data sets, value ranges of predicates are from tens to thousands. According to our experiments, $\epsilon \in [0.01, 0.02]$ is a reasonable choice.

It is also worth noting that all these decisions are heuristic and subject to change for different applications. When applying this approach, one is required to perform some preliminary experiments to decide suitable parameters.

5. RESULTS FOR SIEMENS TEST SUITE

In this section, we present results of our experiments on the Siemens test suite. We present a summary of the results first before presenting the detailed results.

5.1 Result Summary

We compare our approach with the two most closely related work: Liblit05 [21] and Liu05 [22]. The data for Liblit05 and Liu05 are taken from Liu *et al.*'s work on SOBER [22]. Table 1 summaries our results (detailed results are shown in Tables 2 and 3). It shows the numbers of bugs detected by different approaches when a programmer is expected to inspect a certain fraction of the code. For example, along the cause-effect chains chosen by our approach, 38 bugs can be discovered by inspecting no more than 1% of the code in each program. We point out that the percentages for Liblit05 and Liu05 are in terms of nodes in the *program dependence graphs* (PDGs) [13], while we use the number of source lines. This difference should be negligible because a node in PDG roughly corresponds to a statement in code which is usually one line. The row labeled **avg** is the ratio of total code examined over total code size for the row labeled "$\leq 10\%$." This average is the expected code fragment that needs to be inspected to locate a bug. We view this metric as the most suitable for comparing the quality of bug localization algorithms.

The data for Liblit05 and Liu05 are computed from Figure 5(b) in Liu05 [22] when feasible. The two **avg**s for them are approximate lower and upper bounds for their "top-5-selected-predicate-based inspection." Although they may not be absolutely accurate, the actual values should be within these ranges. The data indicate rather clearly that the approach in this paper is the most accurate; it not only discovers more bugs but also incurs significantly less human efforts in locating bugs. Our approach achieves much better detection rates, especially when the expected size of code to inspect is low. The dominance of Liu05 and Libit05 over ours shows when more code is examined (at 20% range), but we do not view this as a drawback of our technique. First, we would still have a much lower average. Second, we would stop code inspection if we do not discover bugs in the chosen cause-effect chains even if we have not reached the code limit (which is natural for manual code inspection). However, Liu *et al.* count the numbers for Liblit05 and Liu05 by breadth-first searching PDGs starting from the locations of selected predicates and do not stop until a bug is discovered or the code limit is reached.

5.2 Detailed Results

We show detailed results in Tables 2 and 3. They are provided here only as reference for those readers who are interested in seeing

% Code Examined	This Paper	Liblit05 [21]	Liu05 [22]
≤ 1 line	11	n/a	n/a
≤ 2 lines	28	n/a	n/a
≤ 5 lines	51	n/a	n/a
≤10 lines	64	n/a	n/a
≤ 1%	38	10	11
≤ 2%	45	24	25
≤ 4%	60	30	40
≤ 6%	67	39	44
≤ 8%	71	43	52
≤ 10%	74	52	68
≤ 20%	81	83	96
avg per bug	1.0%	2.6-5.1 %	3.1-5.3 %

Table 1: Result summary for the Siemens suite. Each entry denotes the number of bugs located if examining certain amount of the code.

such data. For each program, we list its buggy version IDs, number of failed executions of each version, line IDs of bug locations, line IDs of bug-related predicates, and the numbers of lines inspected along cause-effect chains in order to locate the real bugs.

When counting the numbers of lines inspected, we first look at the specific lines corresponding to the selected bug-related predicates, then we start from the head of cause-effect chains until we reach real bug locations. If a chain is a loop or it has several heads, we start from the line which corresponds to the first selected bug-related predicate. In several cases, when a real bug is in the same basic block as the head of a chain and within three lines before the head, it is usually easily visible by human inspectors, and we track the lines backwards to reach the bug faster. In terms of PDG, one can traverse along reversed directed edges three steps or until a merge node is met; the only difference is that we do not track into inter-procedural calls unless a predicate on the chain indicates us to do so. Also, we assume a human inspector can determine whether it is a real bug when a line is presented, as done in previous work [7, 22], to make a fair comparison. When a bug location is beyond any chain reported, we simply leave the line number as blank in Tables 2 and 3.

In most cases, the selected predicates from SVMs and RFs are consistent with each other, and sometime they complement each other. We analyze them together just assuming predicates from SVMs have priorities over those from RFs. We leave as future work to combine different machine learning techniques together (*e.g.*, votes on predicates) to further reduce code needed for manual inspection.

Version IDs with a "@" are the ones whose bugs are located directly by selected predicates. There are 45 such versions. Version IDs with a "*" are the ones whose bugs are contained on the cause-effect chains reported. There are totally 38 stars. Thus, one can detect 83 bugs by examining the chains reported by our approach, which amounts to 1037 source lines in the 132 versions with a total number of around 43,000 source lines.

In addition, "+" signs are used to indicate the effectiveness of our clustering strategy. A version ID has a "+" label if: (1) the clusters generated by our clustering directly contain bugs, while the selected predicates alone do not; or (2) the chains based on the clusters contain bugs, while the chains based the selected predicates alone do not. There are 14 such labels. We notice that most of them correspond to larger numbers of lines requiring inspection, which means that such a clustering strategy may be helpful for discovering bugs spanning large code ranges.

Let us consider a concrete example. The selected predicates for version 1 of `print_token` correspond to the lines 320, 228, 204, 207 and 460 in the code, and they are clustered into four clusters containing many other predicates. The cause-effect analysis, for example, for the clusters for 320, 204 and 207, are shown below with sequential statements and function calls and returns in between and are omitted if it is clear from the original code.

- The cluster for 320 only has one member, which usually means the code around 320 is almost stand alone. It turns out that the code around 320 generates an error token and likely to end the test run.
- 204 and 207 are in the same cluster, which contains a total of 11 predicates showed below. The corresponding cause-effect chain is the third shortest one, after the cluster for 320 and the cluster for 460. After inspecting 88 lines along the three chains (overlapped parts are counted only once), we located the bug on line 227, which is due to a mis-handled switch-case.

```
   Line 94: stream_ptr->stream[...]=='\0'
            is false in get_char;
→ Line 193: !token_found is true in get_token;
→ Line 195: token_ind<80 is true in get_token;
→ Line 460: state<0 is false in next_state;
→ Line 462: base[state]+ch>=0 is true
            in next_state;
→ Line 464: check[base[state]+ch]==state
            is true in next_state;
→ Line 204: next_st == -1 is false in get_token;
→ Line 207: next_st == -2 is false in get_token;
→ Line 210: next_st == -3 is false in get_token;
→ Line 248: call get_actual_token;
→ Line 555: ind>0 is true in get_actual_token;
→ Line 558: ind<token_ind is true
            in get_actual_token;
```

We see that the three chains cross through seven functions and 88 lines out of a total of 18 functions and 402 lines. The bug happens to be one of the worse cases among all bugs that our approach can localize in the Siemens test suite, requiring inspection about 22% of the code.

The overall bug detection rate of our approach, 74 out of 132, compared with 68, the best previously reported result out of 130 [22], by inspecting no more than 10% of the code. Within the 10% limit, our approach averagely requires inspection on about 1% of the code in order to locate one bug, while Liblit05 and Liu05 both require more than 2.6%.

6. RESULTS FOR RHYTHMBOX

6.1 Result Summary

Based on the bug-related predicates and clusters generated by our approach, we were able to localize 5 bugs in `rhythmbox 0.6.4`, after examining less than 1000 lines of the code, out of a total of 56484 lines of code [21]. The causes of three of them are similar to the bugs discovered by Liblit [17, 18]. Two of the bugs have been fixed in recent versions of `rhythmbox`; the code related to another bug has been discarded; the other two are yet to be confirmed by the developers.

It takes several hours for our approach to localize the bugs, including data preprocessing, learning and clustering. It takes additional time, ranging from minutes to a few hours, to perform code inspection to understand the cause-effect chains. We believe this is reasonable for a real, large application that we are not familiar with and that contains unknown bugs.

Ver	Failed #	Bug Location	Detected(SVM)	Detected(RF)	Lines Examined
print_tokens					
1*+	6	226-236	320, 228	204, 207, 460	88
2	48	226	72, 70, 41, 315, 487	463, 315, 117	
3*+	38	235	227	344, 227	88
4	28	in header files	462, 210, 417, 420	460, 204, 195	
5	150	253	420, 417, 561	94, 417, 420	
6	186	in header files	462, 320, 561	204, 314, 193	
7@	28	281	281	561, 279, 281	1
print_tokens2					
1	241	188	219	219, 481, 489	
2*+	250	193	226, 189	189, 226	10
3	34	176	210, 379, 435	209, 379, 210	
4@	333	164	309, 246, 164	483, 491, 479	3
5*	174	386	380	380, 210, 381	7
6@	519	358	358, 354	356, 358	1
7@	208	218	218	218, 217, 210	1
8@	257	225	225	225	1
9@	57	218	218	210, 295, 218	1
10@	174	380	381, 380, 211	380, 295, 381	2
replace					
1@	68	107	107	315, 317, 105	1
2@	37	111	114, 111, 113	317, 315, 105	2
3*	130	494	448, 498, 164	428, 439, 431	38
4@	143	494	498, 494, 182	494, 498, 428	2
5@	271	118	141, 118, 115	317, 315, 182	2
6*	96	315, 319	370, 358, 118	182, 315, 317	5
7@	83	176	209, 176, 79, 82	176, 203, 543	2
8	54	176	514, 491	491, 427, 514	
9@	30	115	118, 115, 141	366, 315, 182	2
10@	23	115	119, 115, 79	367, 318, 316	2
11*	30	116	141, 118, 115	366, 315, 182	18
12	309	15	56, 108, 72	56, 239, 284	
13@	163	500	498, 500, 494	500, 498, 428	2
14	137	370	141, 107, 115	494, 182, 427	
15*+	60	241	498, 494, 514, 419	427, 467, 494	58
16@	83	176	176, 209, 79, 82	176, 217, 55	1
17*	24	75	205, 74, 366	341, 467, 491	71
18@	210	372	141, 337, 182	372, 182, 315	4
19	39	44	470	338, 429, 492	
20*	22	75	74, 205, 366	427, 491, 337	72
21	3	14, 44, 55, 209	79, 209, 205	427, 337, 362	
22@	19	140	140, 493, 163	181, 369, 316	1
23*+	22	74	498, 271, 468	71, 271, 270	6
24*+	170	362	209, 447, 497	182, 341, 490	42
25@	3	362	494, 498, 428	362, 182, 317	4
26@	295	370	141, 354, 105	370, 315, 317	4
27@	263	182	341, 182, 514	209, 182, 341	2
28@	142	176	55, 118, 164	176, 219, 118	4
29@	64	176	164	235, 176, 203	3
30@	284	176	79, 82, 55	176, 219, 203	4
31@	210	370	141, 337, 494, 498	370, 182, 317	4
32	0	115	n/a	n/a	
schedule					
1@	4	105	325, 105, 257	120, 127, 325	2
2*	210	230-231	81, 290, 127	120, 180, 233	13
3*	159	209	105, 211, 233	105, 120, 180	9
4*	294	207	289, 201, 182	324, 180, 201	7
5	37	212-215	120, 228, 246	105, 233, 120	
6@	4	105	325, 105, 257	120, 127, 325	2
7@	27	210, 233	233, 210, 77	233, 123, 210	2
8	31	215	182, 180, 257	102, 120, 81	
9@	23	314	290, 314, 321	321, 290, 77	2
schedule2					
1	65	136	275, 120, 125	275, 28, 292	
2*	31	296	50	209, 291, 292	11
3*	34	293	295	295, 296, 299	8
4*	2	93	75, 60, 147	292, 291, 60	19
5@	32	111	185, 111, 162	293, 111, 210	2
6	7	77	171, 295, 147	291, 292, 211	
7@	31	292	50, 292, 209	292, 209, 291	2
8	67	275	preprocessing failed		
9	0	187	n/a	n/a	
10*+	46	28	275	292, 291, 275	12
tot_info					
1*	158	342	340	340, 336, 55	3
2@	10	85	378, 85, 84	85, 84, 340	2
3	3	75	63, 109, 340	84	
4	33	233	55	385, 381, 364	
5*	29	105	117, 55, 75	174, 162, 248	65
6	46	18	60, 57, 55, 84, 83	364, 340, 85	
7*+	104	378	57, 191, 55	385, 394, 57	58
8*	199	201	55, 200, 196	196, 200, 191	5
9*+	37	106	117, 55, 75	99, 57, 174	52
10	8	301	233, 162, 191	394, 360, 99	
11*	199	198	55	196, 200, 191	5
12	33	177	200, 196, 394	200, 196, 381	
13*+	123	394	57, 378, 385	385, 378, 374	48
14	2	75	63, 109, 340	no selection	
15@	199	200	55	196, 200, 191	3
16@	168	99	55, 352, 109	109, 374, 99	6
17	44	223	55	385, 364, 381	
18	119	308	55	336, 109, 332	
19	89	18	55, 60, 57	84, 336, 340	
20*	88	308	385, 378, 63	84, 336, 83	27
21	115	22	162, 57, 85, 84	84, 85, 381	
22@	23	352	55, 308, 57	57, 352, 308	4
23	71	215	55	385, 381, 336	
tcas is continued on Table 3					

Table 2: Bug localization results for the Siemens test suite.

Ver	Failed #	Bug Location	Detected(SVM)	Detected(RF)	Lines Examined
1@	132	75	93, 127, 75, 126, 128, 133, 135	75, 133, 126	3
2@	69	63	120, 124, 126	63, 124, 118	4
3*	23	120	124, 135, 126	124, 75, 133	3
4@	24	79	79, 133, 128, 126	73, 91, 79	1
5@	10	118	93, 135, 79	79, 118, 127	5
6	12	104	120	73, 91, 133	
7	36	51	133, 128, 126	75, 93, 128	
8	1	53	no selection	126, 63	
9*+	9	89	78, 125, 72, 74	78, 90	13
10	14	105, 111	122, 129, 137	73, 91, 128	
11*+	14	106, 113, 136-144	124, 131, 141	73, 91, 93	13
12@	70	118	135, 79, 118	75, 93, 79	3
13	4	10	120, 124	no selection	
14	50	11	118	126, 124	
15	10	12, 118	93, 124, 135	93, 79	
16	70	50	75, 93, 126, 128, 133	73, 91, 120	
17	35	51	133, 128, 126, 93, 75	75, 93	
18	29	52	120, 124, 127	93, 75	
19	19	53	133, 128, 126, 93, 75	126	
20*+	18	72	127, 97, 91, 93	133, 128, 127	6
21*	16	72	127, 97, 91, 93	128, 75, 126	6
22*+	11	72	93, 127	124	4
23*	41	90	79, 126, 73, 75	79, 91	6
24*	7	90	75, 73, 79	127, 97, 135	5
25@	3	97	79, 126, 97, 127, 135	no selection	3
26	11	118	135, 126, 127, 97, 79	63, 97, 127	
27@	10	118	93, 135, 79	79, 118	4
28	75	63	97, 79, 126	93, 75	
29	18	63	120, 124, 127	124, 118	
30	57	63	120, 124, 127	93, 126, 133	
31*	14	76, 81, 128	95, 75, 135, 130, 76	93, 75	10
32*	2	94, 99, 129	137, 99	no selection	8
33	89	50-53	127, 93, 75	75, 93, 127	
34@	77	124	124, 118, 135	93, 118, 128	1
35	75	63	97, 79, 126	93, 75	
36	123	46	135, 127, 97, 79	97, 127, 135	
37	97	58	127, 93, 75	124, 126, 118	
38	55	27	93, 75, 79, 97	118, 93, 75	
39@	3	97	79, 97, 126	no selection	2
40*	123	75, 126	128, 127, 97, 79	79, 128, 127, 97	8
41@	24	79	79, 133, 128, 126	91, 79, 73	1

Table 3: Bug localization results for `tcas`.

Our experiments also show that the `branches` scheme is the most effective one among the three different instrumentation schemes. This confirms that many defects are only exhibited on certain execution paths which are usually determined by branch conditions. It also confirms our results on the Siemens suite that used only `branches` predicates. This is helpful because it provides hints about what types of bugs are the most common so that we can design more expressive instrumentation predicates for future use.

Our main experiments are done on about $1,711$ executions including 247 failed runs. These runs are already sufficient for us to identify many interesting bug-related predicates, as shown in Table 5. The random forest-based experiments also show the potential of locating bugs with smaller data set, which suggests that perhaps we do not need tens of thousands runs as Liblit05 [21] did. Thus, it may be possible to significantly lower the cost of data collection, predicate selection and clustering.

6.2 Detailed Results

Table 4 shows the performance of one of our main experiments on `rhythmbox`. The time of "preprocessing" includes the time for data format transformation, scaling, and making up missing values (as zeros). The time of "learning" is the time used to establish the classification model (done only once). We attempted to iteratively adjust model parameters and improve the classification accuracy as discussed in Section 3.3. The time of such a process is listed as "adjusting." This is an optional step and did not have much impact on our bug localization results.

For a larger data subset with $3,051$ succeeded and 159 failed executions, SVMs fail to terminate for the `scalar-pairs` instrumentation scheme in about 30 hours. Due to limitations of the evaluation version of RandomForests that we have available, experiments based on RFs are only done for a smaller subset with

Instrumentation Schemes	Number of Predicates	Time (minutes) of Preprocessing	Learning	Adjusting	Clustering	Classification Accuracy
branches	6,863	40	0.2	88	30	99%
returns	25,287	44	0.25	103	770	99%
scalar-pairs	400,185	647	7	4920	n/a	96%

Table 4: Performance of SVM-based learning on a data subset with 1464 **succeeded executions and** 247 **failed executions. The adjusting steps are optional.**

219 succeeded and 87 failed executions for the `branches` scheme and finishes in 20 minutes. They still generated some interesting bug-related predicates, compared with the experiments on SVMs for a much larger data set (Table 5).

Now we explain some of the bug-related predicates shown in Table 5, and their corresponding clusters and the derived cause-effect chains. The results are obtained during the same experiment as shown in Table 4.

Because `rhythmbox` is a multiple-threaded application, it uses many function pointers to handle signals, and our basic CFG analysis has not taken thread switching and control flows via function pointers into account, so the cause-effect chains reported are often not consecutive or directly expose bugs. However, predicate selection and clustering do give us hints about the relationships among different parts of the application and where to start code inspection. We are new to `rhythmbox`, `GTK+` and `GNOME` related programming, and it takes us ranging from minutes to hours to understand each potential bug.

- The first cluster contains the first three, the fifth and the sixth predicates from the `branches` scheme.

```
Line 271: monkey_media_is_alive()==FALSE is
       not executed in monkey_media_shutdown.
Line 287: i<impl_array->len is not executed
       in monkey_media_shutdown.
Line 41: global_gconf_client==NULL is
       not executed in global_client_free.
Line 1740:selected_entry!=view->priv->selected_entry
       is not executed in
           rb_entry_view_selection_changed_cb.
Line 1815: !player->priv->url is not executed
       in rb_song_display_box_size_request
```

  The failed runs in our experiments are all crashes, and normal code for exit is not executed in such runs. Thus, it is natural that the first three exit-related predicates are not executed. The other two predicates are used to handle windows events. The fact that they are not executed means failures happened before or just on them. Although our approach does not expose the failure causes directly, it gives hints to conduct later analysis. We located a race condition in the fucntion `monkey_media_shutdown` where a file-scope variable `alive` is used as a guard for the `monkey media player`. However, no protection is enforced on `alive` and the player could be shut down more than once and thus cause crashes on exit. [2]

- The fourth predicate from both the `branches` and the `return` schemes are in fact the same, and is in a stand alone cluster. From our experiences, such a singleton cluster usually indicates that the predicate is bug-related and is worth careful inspection. Admittedly, it is more difficult to backtrack than clusters with more than one predicates, because of a lack of information for choosing cause-effect chains.

  CBI has also identified `monkey_media_player_get_uri ==NULL` in the function `info_available_cb` as a crash indicator and explained its cause [18]. Our approach ranks the predicate much higher than CBI. The bug involves many callback functions, and we need an accurate global CFG to automatically track the origins of such bugs.

- The cluster containing the seventh predicate in the `branches` scheme has 36 predicates, across 20 functions in eight files. The cause-effect chains are not consecutive due to the limitation of our current CFG analysis. However, a highlighted piece of the chain shows the following suspicious execution path in `rb_shell_player_set_playing_source_internal`:

```
...
→ Line 1537: player->priv->source!=NULL is false;
→ Line 1551: source!=NULL is false;
→ Line 1566: source==NULL is true and call
             rb_shell_player_stop(player);
→ Line 1581: monkey_media_player_playing(player...)
             is false
```

  The chain shows a crash after changing playing source to a should-not `NULL`. Based on a more careful examination and our knowledge from previous bug reports [17, 18], a possible cause is the signal notification via `g_signal_connect` between line 1551 and 1566 sometimes attempts to operate on an already freed object. Newer versions have changed it to `g_signal_connect_object` to ensure the object is alive.

  The chain also shows the advantage of our clustering strategy: collect as many related predicates as possible to facilitate inspections on cause-effect chains. This is particularly helpful for predicates influenced by similar bugs spanning a wide code range in the programs. Without clustering, we could not even know where to inspect given just a single predicate.

- We also get another singleton cluster containing the eighth predicate from both the `branches` and the `returns` scheme.

```
Line 416: view->priv->change_sig_queued is true
        in rb_entry_view_finalize.

static void rb_entry_view_finalize (...) {
  ...
  if (view->priv->change_sig_queued)
    g_source_remove (view->priv->change_sig_id);
  ...
}
```

  The predicate indicates crashes when the `g_source_remove` is called for releasing resources, which means `change_sig_id` may be a dangling resource ID. Liblit [17] has also identified another call to `g_source_remove` as a crash indicator. Employing basic dataflow analysis to trace the origins of the values of `change_sig_id`, we found that `change_sig_queued` could become a race condition and cause inconsistency between `change_sig_id` and itself and suggested a fix to the developers.

- As another instance of singleton clusters, the seventh predicate in the `returns` scheme leads us to the following code:

[2] Newer versions of `rhythmbox` restrict the functionality of `monkey media` to reading and writing metadata only and have abandoned the use of `alive`, due to this problem and some other additional reasons.

Instrumentation Schemes and Selected Predicates				
	SVMs			RFs
#	**branches**	**returns**	**scalar-pairs**	**branches**
1	`global_gconf_client==NULL`	`g_ptr_array_free>0`	`i==2`	`g_threads_got_initialized`
2	`i<impl_array->len`	`g_type_check_instance_cast>0`	`...`	`children`
3	`monkey_media_is_alive()==0`	`monkey_media_is_alive>0`	`alive==0`	`gdk_threads_mutex`
4	`!monkey_media_player_get_uri`	`monkey_media_player_get_uri==0`	`...`	`child_requisition.height`
5	`selected_entry!= view->priv->selected_entry`	`g_strdup>0`	`global_gconf_client ==0`	`size!=-1`
6	`!player->priv->url`	`rb_entry_view_get_entry_contained>0`	`...`	`monkey_media_is_alive()==FALSE`
7	`!rb_entry_view_get_entry_contained`	`rhythmdb_query_model_entry_to_iter >0`	`data->shell->priv ->play_queued<1287`	`global_gconf_client==FALSE`
8	`view->priv->change_sig_queued`	`g_source_remove>0`	`cc>cc`	`GTK_WIDGET_VISIBLE(child->widget)`
9	`g_threads_got_initialized`	`rb_entry_view_get_playing_entry>0`	`changed==callback_runs`	`i<impl_array->len`

Table 5: Bug-related predicates for `rhythmbox`. Results for SVMs are from experiments on 1464 succeeded and 247 failed runs. Results for RFs are from 219 succeeded and 87 failed runs, and only for the branch scheme, due to limitations of the tools that we have available.

```
void rb_entry_view_select_none(RBEntryView *view) {
    view->priv->selection_lock = TRUE;
    ...
    view->priv->selection_lock = FALSE;
}
void rb_entry_view_select_entry (...) {
    ...
    view->priv->selection_lock = TRUE;
    rb_entry_view_select_none (view);
    if( rhythmdb_query_model_entry_to_iter (...) )
        ... /* change selections */
    view->priv->selection_lock = FALSE;
}
```

Note that the `selection_lock` is always `FALSE` in the `if` branch and will cause race conditions in functions that access the `view` elsewhere.

7. DISCUSSION

From the experiments we see that machine learning techniques are capable of localizing bugs in many cases, but not every case. Several factors have impact on the effectiveness of machine learning-based bug localization. One of the most crucial one concerns instrumentation schemes. Most bugs that we discovered are related with failure-indicating branch conditions, which attributes to the branch instrumentation schemes. Many bugs due to internal data errors (*e.g.*, version 12 in `replace`, version 7 in `tcas`, and version 6 in `tot_info`) are not easy to be localized. This drawback is fundamental to feature-based machine learning whose effectiveness on discovering knowledge is confined to features used to depict knowledge. To make instrumentation predicates more expressive and target different kinds of bugs may improve the effectiveness of bug localization. We can also apply advanced program analysis to select appropriate predicates for instrumentation, decreasing the total number of instrumentation predicates while still retaining their effectiveness.

The second factor concerns the types of bugs. For some types of bugs, it is difficult to instrument or to determine whether they are real bugs. For example, version 1 in `print_tokens2` used wrong logic in handling tokens, and version 10 in `tot_info` has a a declaration with wrong precision. Many such bugs escaped our detection, and one needs to design special treatments for these different types of bugs.

The third factor concerns the size of a data set, and particularly, the number of failed runs. When the number of failed runs is not enough, *e.g.*, version 8 in `tcas` and version 14 in `tot_info`, our approach cannot identify useful predicates, and the code range for inspection may be substantial. We believe, though, that this factor is not a major concern for an instrumentation infrastructure such as `CBI`, because in real deployment, the amount of data should be sufficient.

The above strategies are mainly for inspecting control flow paths. We can also track data flows and discover additional bugs. For example, our approach can already locate the `if` branch in the following code segment from version 36 of `tcas`, which is closely related with bugs. However, we have not tracked any data flows in the Siemens test suite and missed the bug:

```
#define DOWNWARD_RA 1 /* should be 2 */
... /* far away */
else if (need_downward_RA)
    alt_sep = DOWNWARD_RA;
```

We also see that the necessity of data-flow analysis from the experiments on `rhythmbox`. When a program is data- or event-driven, it utilizes many function pointers to implement call back mechanisms, which makes generating control flow graphs and choosing cause-failure chains more difficult. Thus, it would be beneficial to integrate these program analysis techniques for both instrumentation and inspection.

As an alternative of choosing cause-failure chains, we can try to restore a failed execution, instead of a naïve control flow path. It is possible because we could obtain a portion of the program states after predicate selection and clustering, based on the values of these predicates when the program fails, and techniques such as *postmortem symbolic evaluation* [23] may be applied to restore an execution leading to the failures. The restored failed execution, including highlighted failure-related predicates, would be more accurate and informative for manual inspection.

8. RELATED WORK

In this section, we survey additional closely related work that has not yet been covered. Brun and Ernst [4] use Daikon, a dynamic invariant detection tool [9], to generate properties of programs. A rich set of properties from Daikon are used as features for machine learning algorithms. In particular, they use decision trees and support vector machines to identify fault-revealing properties. Their approach requires different versions of the same program, and one of those is assumed free of faults. All properties from this version are also assumed to be non-fault-revealing and thus can be used as references in determining whether other properties are fault-revealing or not. In our framework, the experiments are based on successful and failed executions of the same program, we do not assume the existence of a correct version of the pro-

gram. Properties used by Brun and Ernst are more expressive than predicates from CBI. Although we can use the simple predicates to locate many bugs, it would be interesting to enhance CBI with more expressive properties (such as related to heap data structures) to locate more classes of bugs.

Also related is Dickinson *et al.*'s work to find failed executions from a set of program executions [8]. They assume that failed executions have unusual profiles and can be identified from other executions to facilitate the revealing of failures. Given a set of execution profiles induced by a set of potential test cases, they utilize *cluster filtering* strategies to group similar executions together and perform sampling evaluation on clusters to find failed executions. Thus, their techniques are used to filter out succeeded test cases and reduce the number of executions requiring (manual) inspection for bugs, while ours filter out non-bug-relevant code and reduce the amount of code that is required for manual inspection. Another difference between cluster filtering and our approach is that the former requires full execution profiles and is originally used for in-house testing, while our study works on summarized profiles.

Renieris *et al.* [25] employ an argumentation of the Hamming distance metric to find a succeeded run that resembles a given failed run from a large set of succeeded runs. Then different program features, such as the times a basic block is executed in a run, between the failed and the selected succeeded runs are flagged for human inspection. Their technique selects features based on one failed run and one succeeded run only, while our study exploits the complete set of runs and selects features using machine learning techniques. Their distance metric is similar to ours in the clustering strategy, but with a different purpose: they use it to look for the nearest neighbors of failed runs to carry out feature selection; we use it to look for neighbors of features and discover relations among features to improve the effectiveness of feature selection.

Our work is also related to *delta debugging* [7, 29, 30] that helps minimize failure-inducing inputs and locate not only failures, but also failure *cause transitions*–moments when new relevant variables become failure-related or cause failures. Their technique requires detailed program states of one failed run and one succeeded run; the states from the two runs are grafted into each other in a binary search fashion to decide which states are cause transitions. Such a technique needs to execute programs dynamically, while our clustering and CFG analysis are done statically and use profiled execution states.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel application of advanced machine learning techniques to accurately isolate cause-effect chains of bugs. We have validated our technique on instrumented code with the CBI infrastructure, including the Siemens test suite and `rhythmbox`. Experiments show that our technique is very effective in locating cause-effect chains. Our technique is able to isolate more bugs than previous approaches and at the same time only requires a programmer to manually inspect a small fraction of the code necessary for other approaches. We believe this is a promising approach and an important step forward to realizing our ultimate goal of automated debugging.

There are also some interesting directions for future work. Some steps in our current implementation are not completely automated. For example, to obtain a model with high classification accuracy, we may need to manually adjust model parameters; and to locate cause-failure chains, we may need to construct global high-level CFGs and manually adjust them according to different values of predicates. As future work, we plan to build our heuristics for performing these tasks into a complete automated system to make our approach more readily applicable.

10. REFERENCES

[1] *Aristotle Analysis System – Siemens Programs, HR Variants*. http://www.cc.gatech.edu/aristotle/Tools/subjects/, 2005.

[2] P. Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[3] L. Breiman. Random forests. *Machine Learning*, 45(1):5, Oct. 2001.

[4] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 480–490, 2004.

[5] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[6] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*. `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[7] H. Cleve and A. Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, pages 342–351, 2005.

[8] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *International Conference on Software Engineering*, pages 339–348, 2001.

[9] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[10] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[11] N. Grira, M. Crucianu, and N. Boujemaa. Unsupervised and semi-supervised clustering: a brief survey. In *A Review of Machine Learning Techniques for Processing Multimedia Content, Report of the MUSCLE European Network of Excellence (6th Framework Programme)*, 2005.

[12] M. Heiler, D. Cremers, and C. Schnörr. Efficient feature subset selection for support vector machines. Technical Report 21/2001, Computer Science Series, University of Mannheim, 2001.

[13] S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering*, pages 392–411, 1992.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[15] L. Jiang and Z. Su. Automatic isolation of cause-effect chains with machine learning. Technical report, University of California, Davis, Dec. 2005.

[16] E. Lehmann. *Testing Statistical Hypotheses*. Springer, second edition, 1997.

[17] B. Liblit. *Bug 137460: dangling timeout event source ID causes crashes*. `http://bugzilla.gnome.org/show_bug.cgi?id=137460`.

[18] B. Liblit. *Bug 137834: dangling RBShellPlayer callbacks cause crashes*. `http://bugzilla.gnome.org/show_bug.cgi?id=137834`.

[19] B. Liblit. *The Cooperative Bug Isolation Project*. `http://www.cs.wisc.edu/cbi/`.

[20] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[21] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 12–15 2005.

[22] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. In *ESEC/FSE 2005*, 2005. To appear.

[23] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *FSE 2004*, 2004.

[24] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[25] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, 2003.

[26] Rhythmbox. `http://www.gnome.org/projects/rhythmbox/`.

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

[28] Salford Systems Inc. *RandomForests*^{TM}. `http://www.salford-systems.com/`, 2005.

[29] A. Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27(6):1–10, 2002.

[30] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[31] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems 16*. 2004.