An Empirical Study on Classification Methods for Alarms from a Bug-Finding Static C Analyzer

Kwangkeun Yi†, Yongdai Kim‡, Jaehwang Kim†, Hosik Choi‡, Jaeho Shin†

†School of Computer Science and Engineering ‡Department of Statistics
Seoul National University
April 5, 2006

1 Motivation

The key application for static analysis is automatic bug-finding. Given the program source, static analyzer computes an approximation of dynamic program states occurring at each program point, and reports possible bugs by examining the approximate states.

From such static bug-finding analysis, false alarms are inevitable. Because static analysis is done at compile-time, exact computation of the program’s run-time states is impossible. Hence some approximation must be involved, so that the detected bugs can contain some false positives. Methodologies such as the abstract interpretation framework[4, 5, 6] counsel us to design a sound static analyzer. The soundness criterion exacerbates the false alarm problem, because the analysis must err on the pessimistic side whenever in doubt.

Reducing the number of false alarms has always been a big challenge in static analysis design. Controlling the approximation level of the analysis will work, but not very effectively. It is clear that using a less approximate analysis can give more precise results, but practically, relying solely on this approach will soon hit an unacceptable analysis cost. Furthermore, if the analyzer must handle an unlimited set of input programs, there will always be a program that fools the analyzer. User annotation in source codes can be effective, yet is always less desirable than full automation. Worse still, the analyzer blindly repairing its accuracy based on annotations renders the approach vulnerable to annotation bugs. Another approach to handling false alarms is to equip the analyzer with all possible techniques for accuracy improvement and let the user choose a suitable combination of techniques for the programs at hand. The library of techniques must be extensive enough to specialize the analyzer for as wide spectrum of the input programs as possible. This approach requires the user to have an inside knowledge of the static analysis techniques, to allow them to decide how to control false alarms.

A promising approach orthogonal to the aforementioned techniques is statistical post-analysis. Given the reported alarms, classification methods compute a “probability” of each alarm being true. We use these probabilities to grade the alarms, so that the user can check highly probable errors first.

One natural question is, which among the many classification methods would be most effective in classifying alarms from bug-finding static analyzers? In this article, we report our experimental results. We attached classifiers to our C analyzer known as Airac [10], which statically detects buffer-overrun errors in C programs. We considered eight classification methods that have been developed in statistics and machine learning: naïve Bayes [9], logistic regression [9], Lasso [11] with logistic regression, classification tree bagging [2], classification tree boosting [8], Breiman’s random forest implementation of ensembles of decision trees [3], and support vector machines [12] with linear and Gaussian kernels.
Table 1: Airac, a realistic bug-finding C analyzer: analysis speed and accuracy

<table>
<thead>
<tr>
<th>Software</th>
<th>#Lines</th>
<th>Time (sec)</th>
<th>#Alarms</th>
<th>#Real bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux kernel version 2.6.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmax302.c</td>
<td>246</td>
<td>0.28s</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>xfrm.user.c</td>
<td>1,201</td>
<td>45.07s</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2.6.4.usb-midi.c</td>
<td>2,206</td>
<td>91.32s</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>atkbd.c</td>
<td>811</td>
<td>1.99s</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>keyboard.c</td>
<td>1,256</td>
<td>3.36s</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>af/inet.c</td>
<td>1,273</td>
<td>1.17s</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>eata/pio.c</td>
<td>984</td>
<td>7.50s</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>cdc-acm.c</td>
<td>849</td>
<td>3.98s</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ip6/output.c</td>
<td>1,110</td>
<td>1.53s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mptbase.c</td>
<td>6,158</td>
<td>0.79s</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>aty128fb.c</td>
<td>2,466</td>
<td>0.32s</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GNU S/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tar-1.13</td>
<td>20,258</td>
<td>576.79s</td>
<td>66</td>
<td>1</td>
</tr>
<tr>
<td>bison-1.875</td>
<td>25,907</td>
<td>809.35s</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>sed-4.0.8</td>
<td>6,053</td>
<td>1,154.32s</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>gzip-1.2.4a</td>
<td>7,327</td>
<td>794.31s</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>grep-2.5.1</td>
<td>9,297</td>
<td>603.58s</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

2 Airac, a Bug-Finding C Analyzer

Airac [10] is a sound static program analyzer that detects all buffer overrun errors in C programs. A buffer overrun error happens in C programs when an allocated memory is read or written outside the memory range. Airac’s sound design is based on the abstract interpretation framework [4, 5, 6].

Airac is an industrial-strength analyzer, which has been successfully used to detect buffer overrun errors in GNU softwares, Linux kernel sources, and commercial programs. It has been used in industry for more than a year. Table 1 shows typical performance statistics for Airac. “#Lines” is the number of lines in the C source files before they are preprocessed. “Time” is the user CPU time in seconds. “#Alarms” is the number of buffer-access expressions that may overrun their target buffers. “#Real Bugs” is the number of buffer accesses that are confirmed to be able to cause real overruns. Each analysis was done in a Linux system with a Pentium4 3.2GHz CPU and 4GB of RAM.

3 Experiments on False-Alarm Classification Methods

We considered eight classification methods. For a probability modeling technique, we used naïve Bayes [9]. For linear regression techniques, we used logistic regression [9] and Lasso [11]. For machine learning techniques, we used three approaches built around classification trees – bagging [2], boosting [8] and random forest [3], a form of ensemble learner – and two support vector machines [12] (one with a linear kernel, the other with a Gaussian kernel). In our experiments, we use the R system’s [1](version 2.2.1) packages for all classification methods.

3.1 Experiments

First, we collected sample alarms from Airac, by applying it to three varieties of programs: device drivers and modules of Linux kernel 2.6.4 (including those in Table 1), several programs used in algorithm textbooks for demonstration, and short programs which were arbitrarily written to test Airac. We then manually inspected the alarms and classified them as either true or false. We take this as the correct classification. The total number of the sample alarms is 332, of which 269 are false alarms and 63 are true. Airac also extracted “symptoms” (see Section 3.2) for each of the alarms.

Then, we trialled the eight classification methods with the above sample alarms. For each experiment, we divided the sample alarms into two groups: a training and a test set. The training set is used to derive a classifier from each classification method. The test set is used to test the derived classifier. We fixed the training set size as 232, and the test set size at 100, (a 7:3 ratio). We sampled the test and training sets 100 times, using stratified sampling (i.e., we uniformly randomly sampled 19 true and 81 false alarms respectively, reflecting the class distributions in the data. These became the test set, the remaining instances forming
the training set). For each sampling, we ran an experiment, consisting of applying all eight classification methods to the training set, and measuring its performance on the test set. We repeated the experiment 100 times, once for each training / test set partitioning. We checked the test-set classification results against the correct classification, to measure the classification method's effectiveness.

The classification methods compute grades for the alarms. Grades are scores in $[0, 1]$. Each alarm's score from the logistic regression and Lasso methods is the probability of the alarm being true, while the scores from other methods indicate the relative strength with which the alarm should be true, in comparison with other alarms. The classification is determined by fixing a threshold grade: alarms with higher grades than the threshold are classified to be true, and alarms with lower grades to be false. The effectiveness of classification methods is determined by how much mis-classification is done while we vary the threshold grades.

3.2 Symptoms

For each alarm, Airac also provides its symptoms that are used as the attributes in the classification methods. The set of symptoms that Airac examines are those reported in [10], namely 18 symptoms that may influence the analysis accuracy. The symptoms, which are all common sense, can be classified into three types: syntactic symptoms, semantics symptoms, and symptoms of alarms themselves. Each alarm from Airac consists of the location of a buffer-overrun expression, its target buffer size, and the overrun index value. Syntactic symptoms describe the syntactic context around the alarmed expressions, e.g., whether the alarmed expression is inside a loop or not. Semantic symptoms reflect analysis operations, whose application during the analysis influence the analysis’ accuracy – for example, whether an inevitable approximation is later refined (by, for example, the “narrowing” operator[6]). Symptoms of the alarms themselves are direct attributes of the alarms – e.g. whether the estimated overrun index is an infinite integer (which strongly suggests that the analysis erred too much).

From a classification perspective, it is noteworthy that the symptoms can be grouped into several highly correlated subsets. For example, an alarm inside a loop (a syntactic symptom) is very likely to have an infinite integer (an alarm symptom) as the buffer access index. Because of this correlation, when we tried logistic regression, which is known to be ineffective for correlated attributes, we used the well-known stepwise-forward-selection technique for selecting only those symptoms with little correlation. For other classification methods, we used all the 18 symptoms without any artificial selection.

3.3 Results

Our experimental results with the eight methods are shown in Figure 1 and Figure 2. Each histogram in Figure 1 shows the number of true and false alarms (shown respectively as positive and negative bars) that received the same grade. From the histograms, relatively weak methods stand out. The bagging, naïve Bayes, and logistic regression methods are weak. They rank many true alarms very low, with some sizable positive peaks standing in low ranks.

For an exact comparison, we refer to the Receiver Operating Characteristic (ROC) curves [7] shown in Figure 2. ROC plots show, over the range of the 100 threshold grades, how many true alarms are included (y-axis) versus how many false alarms are included (x-axis). Thus the lower the mis-classification error, the closer the plot moves to the upper left corner (i.e., the more similar to a Γ-shape). The area under the ROC curve (AUC) is thus an overall measure of a forecast’s accuracy. AUC=1 indicates a perfect forecast, while AUC=0.5 indicates a random forecast. The AUC calculation for each ROC curve is shown in Table 2. By this measure, the effective classification methods are clearly exposed: boosting, random forest, and support vector machine methods are the most effective.

From a close examination of the ROC data, the random forest is the most effective in excluding false alarms in high grades. Suppose the user sees higher-ranked alarms earlier.
Then, by the time 50% (950) of the 1900 true alarms have been seen, only 0.32% (26) of the 8100 false alarms have been mixed with them. It is particularly impressive that no false alarm are seen until 22.58% (429) of the 1900 true alarms have been seen. Meanwhile, for the boosting method, at the 50% true alarm stage, 1.21% (98) of the false alarms will also have been seen, and the first false alarm is seen when only 1.10% (21) of the true alarms have been seen, no false alarms.

On the other hand, boosting is slightly better than random forest at excluding true alarms from low grades. For random forest, at the grade threshold by which 50% (4050) of the 8100 false alarms are filtered out, 5.10% (97) of the 1900 true alarms are excluded; for boosting, this number is only 4.47% (86). Thus the preference between boosting and random forest depends on the relative importance of missing true alarms, and of checking false alarms.

4 Conclusion

Random forest [3] and boosting [8] generated the most accurate classifications, among the eight classification methods trialled, in classifying alarms from a bug-finding static C analyzer Airac [10]. The characteristics of symptoms (attributes used in the classification methods) is that they can be grouped into highly correlated subsets.

Although improving the accuracy of the static analysis is the orthodox approach to increasing the effectiveness of bug-finding static analyses, it appears there is value in applying statistical post-analyses. We believe that statistical classification is a relatively easy and effective way to improve the usability of bug-finding static analyses. From our experiments, we can suggest random forest [3] and boosting [8] as the best choices for classifying alarms from bug-finding analyzers, at least when some symptoms are correlated.

References

Figure 1: Alarm distributions of the eight classification methods: the numbers of true and false alarms (respectively as positive and negative bars) at each grade (rank).
Figure 2: ROC (Receiver Operating Characteristic) curves of the eight classification methods. X-axis: #false alarms / total #false alarms. Y-axis: #true alarms / total #true alarms.


