Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 102 (2007) 118–123
Information Processing Letters
www.elsevier.com/locate/ipl

An empirical study on classification methods for alarms from a bug-finding static C analyzer ☆

Kwangkeun Yi *, Hosik Choi, Jaehwang Kim, Yongdai Kim

Seoul National University, Republic of Korea

Received 24 June 2005; received in revised form 3 February 2006; accepted 10 November 2006

Available online 1 December 2006

Communicated by G. Morrisett

Keywords: Static analysis; Abstract interpretation; Statistical post analysis; Classification methods; Program correctness

1. Motivation

A key application for static analysis is automatic bug-finding. Given the program source, a 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 [6–8] counsel us to design a correct (conservative) static analyzer. The correctness criterion exacerbates the false alarm problem, because whenever in doubt the analysis must err on the pessimistic side.

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 code can be effective, yet is always less desirable than full automation. Worse still, blindly using annotations to repair the accuracy of the analyzer renders the approach vulnerable to annotation mistakes. 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

☆ This work was partially supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by IT Leading R&D Support Project of Korea Ministry of Information and Communication, by Korea Research Foundation grant KRF-2003-041-D00528, by Microsoft Asia, and by Korea National Security Research Institute.

* Corresponding author.
E-mail address: kwang@ropas.snu.ac.kr (K. Yi).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2006.11.004

the reported alarms, classification methods compute a "strength" of each alarm being true. We use these quantities to rank the alarms, so that the user can check highly probable errors first. The practicality of this approach has been demonstrated in various settings [5,13].

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 conservative C analyzer Airac [12] which statically detects buffer-overrun errors in C programs. The classifiers are based on features extracted from the bug reports. We train the classifiers on the features for a set of inspected reports, and then evaluate the classifiers on a reserved test set to represent uninspected errors.

2. Airac, a bug-finding C analyzer

Airac [12] is an abstract interpreter [6–8] that detects buffer overrun errors in C programs. A buffer overrun error happens in C programs when allocated memory is read or written outside its valid memory range. Airac does an interprocedural analysis, covering almost all C features (including dynamic allocations, aliases, gotos/breaks, function pointers, and aliases). It chases all accesses to both dynamically and statically allocated buffers. Airac is an industrial-strength analyzer, which has been successfully used to detect buffer overrun errors in GNU software, Linux kernel sources, and commercial programs. It has been used in industry for more than a year. Upon analysis completion, Airac reports positions of errors (alarms) in the input C source, along with symptoms that may signify either the analysis precision or imprecision. These symptoms are later used by the classifiers to rank the alarms.

3. Experiments on false-alarm classification methods

We considered eight classification methods. For a probability modeling technique, we used naïve Bayes [11]. For linear regression techniques, we used logistic regression [11] and Lasso [16]. For machine learning techniques, we used three approaches built around classification trees—bagging [3], boosting [10] and random forest [4], a form of ensemble learner—and two support vector machines [17] (one with a linear kernel, the other with a Gaussian kernel). In our experiments, we use the R system [1] for all classification methods.

3.1. Experiments

First, we collected sample alarms by running Airac on three varieties of programs: 36 files (device drivers and modules) from the Linux kernel 2.6.4 sources, 12 programs in algorithm textbooks, and 10 short programs which were arbitrarily written to test Airac. The total number of the alarms is 332. We manually inspected and classified the alarms as either true or false. Among 332 alarms, 269 are false alarms and 63 are true. We take this as the correct classification. 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. Our experiments are done as follows:

(1) We randomly divide the 332 alarms into two sets: a training set of 232 alarms and the remaining test set of 100 alarms.
(2) We use the 232 alarms of the training set to derive 8 classifiers, one for each method.
(3) We measure the performances of derived 8 classifiers with the 100 alarms of the test set. We checked the test-set classification results against the correct classification.
(4) We repeat the loop of steps (1)–(3) for 100 times in order to avoid, if any, an accidental bias in a division. We simply sum the results over 100 different divisions of the training and the testing sets.

For step (1), we employ a stratified sampling to "randomly" select 100 alarms for the test set: we randomly sample (without replacement) 19 true and 81 false alarms from the total set of 332 sample alarms. The remaining 232 constitute the training set. This sampling method preserves in the test set the ratio of true to false alarms that appears in the entire set of alarms.

The classification methods compute scores for the alarms. Scores are real numbers in [0, 1]. For the logistic regression and Lasso methods, an alarm score is the probability of the alarm being true, while for the other methods the score indicates the relative strength with which the alarm should be true in comparison with other alarms.

The effectiveness of classification methods is determined by how much mis-classification is done while we vary the threshold score. A mis-classification happens when a true (resp. false) alarm has a lower (resp. higher) score than the threshold score.

Scores are employed by Airac to rank alarms from most-to-least probable to be real bugs, or to filter alarms

whose scores are less than a threshold. Such a filtering threshold is determined by a user-provided ratio of the risk of silencing true alarms (false negatives) to that of emitting false alarms [12].

3.2. Symptoms

Each alarm from Airac consists of the location of a buffer-overrun expression, its target buffer size as an integer interval, and the overrun index value also as an integer interval.

For each alarm, Airac also provides its symptoms to be used as attributes in the classification methods. Airac examines in total 20 binary symptoms. Each symptom indicates a situation that may contribute to either the analysis precision or imprecision.

Each symptom indicates the analysis precision or imprecision, and they were are largely derived from observation accrued in the deployment of Airac. Moreover, symptoms are selected without concern for any correlation between them.

The 20 symptoms can be classified into three classes: 12 syntactic symptoms, 5 semantics symptoms, and 3 result symptoms.

• Syntactic symptoms describe the syntactic context around the alarmed expressions. They indicate whether an alarmed expression occurs at a place where the analysis accuracy is readily compromised for analysis termination.

Airac examines the following 12 syntactic symptoms inside the function body that contains an alarmed expression:

AfterLoop AfterBranch AfterReturn CallFunc
CallFuncPtr InLoopCond InBranchCond
InFunParam InNestedFunParam InRightOfAnd
InNestedLoopBody$N$ InNestedBranchBody$N$

AfterLoop and AfterBranch are, respectively, turned on when loops and branches appear before the alarmed expressions. These symptoms are for false alarms; loop and branch can decrease analysis accuracy due to the join operations at their flow-join points. AfterReturn is on when a return statement precedes the alarmed expression. This symptom is for false alarms; while the return commands in the middle of function body are often for exiting on erroneous cases, static analyzers can blindly propagate such erroneous cases to unrelated buffer access expressions. CallFunc and CallFuncPtr are on when a function call (a regular call or a call via a function pointer) precedes in a path an alarmed expression. These are for false alarms because our analyzer is context-insensitive during inter-procedural analysis. InLoopCond and InBranchCond are on when alarms are inside the condition expressions, and InFuncParam and InNestedFuncParam are on when alarms occur inside function's actual parameter expressions. These four symptoms are for true alarms because it is likely that expressions in those contexts are more carefully checked by programmers than expressions in other contexts. InRightOfAnd is for alarms in the right-hand side (rhs) of the logical-and && operator. This symptom is for false alarms because C's short-circuit semantics can skip executing the && operator's rhs expressions. InNestedLoopBody$N$ and InNestedBranchBody$N$ are for true alarms because programmer are easy to mistake inside nested loops and branches. Since we found that simple nested structures were common in both true and false alarms, we refined symptoms by their nesting depth $N = 0, 1, 2, 3,$ or >3.

• Semantic symptoms reflect analysis operations whose application during the analysis influence the analysis' accuracy—e.g., whether an inevitable approximation is later refined using the "narrowing" operator [8].

Airac examines the following 5 semantic symptoms during analysis:

Join$N$ Prune FailPrune FailNarrow InStructure

The number of join operations before the alarmed expressions affects the analysis accuracy. This situation is captured by symptom Join$N$. N is the number of join operations at the nearest control-join point before an alarmed expression. N ranges over $\{1, \ldots, 10, >10\}$. The context pruning (an operation of refining an analyzed program state to an if-branch condition) as well as the narrowing operations are factors that influence analysis accuracy. FailPrune and FailNarrow are on when those operations fail to refine an analyzed program state. Prune is on when the pruning succeeded. InStructure is on when the target buffers are pointed to from some data structures (e.g., record fields). This symptom is for true alarms because such complicated use of the target buffers are likely to be confused.

• Result symptoms are direct attributes of the alarms —e.g., whether the estimated buffer index interval has an infinite integer, which strongly suggests that the analysis erred too much.

Airac examines 3 result symptoms:

TopIndex HalfInfiniteIndex FiniteIndex

If an estimated buffer index is the whole integer interval (the top element of the analysis' lattice) is likely to be a false alarm (TopIndex) because it is likely to have

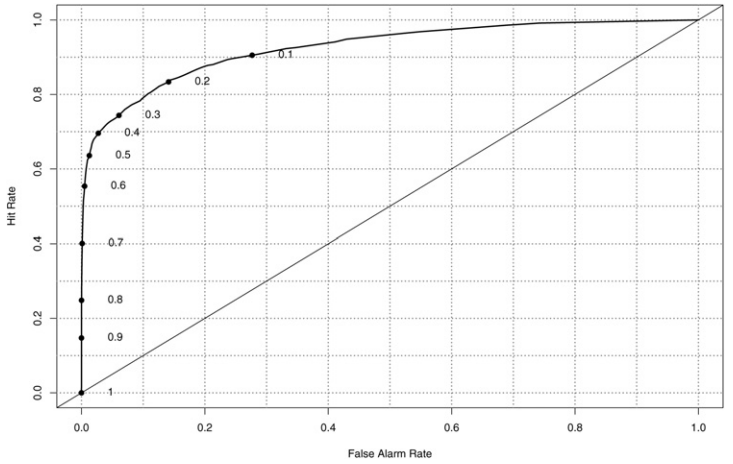

Fig. 1. Example ROC (Receiver Operating Characteristic) curve. X-axis: #false alarms/total #false alarms. Y-axis: #true alarms/total #true alarms. Result from random forest method in our experiment.

erred too much. HalfInfiniteIndex is on when an index interval is half-infinite like $[1, \infty]$. Conversely, buffer indices with exact boundaries (FiniteIndex) strongly suggest true alarms.

From a classification perspective, two things are noteworthy. One is about the generality of these symptoms: the symptoms' three classes would be common across semantics-based static analyzers. Another thing is that some symptoms are correlated. For example, an alarm inside a loop (symptom InNestedLoopBody$N$) is likely to have an infinite buffer access index (symptom HalfInfiniteIndex). 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 symptoms without any artificial selection.

3.3. Results

Each classification method's effectiveness can be visualized by its Receiver Operating Characteristic (ROC) curve [9]. The ROC curve depicts the fractions of included false alarms (x-axis) and true alarms (y-axis). The numbers on the curve are the threshold scores. 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). As an example ROC curve, Fig. 1 shows that of the random forest method in our experiment. Numbers on the ROC curve are threshold scores.

An overall measure of a forecast's accuracy is the area under the ROC curve (AUC). AUC $= 1$ indicates a perfect forecast, while AUC $= 0.5$ indicates a random forecast. The AUC value for each ROC curve is shown in Fig. 2. By this measure, overall effectiveness of the classification methods are exposed: boosting, random forest, and Support Vector Machine (SVM) methods are the most effective.

Method	AUC	Method	AUC
boosting	0.9290	Lasso	0.9095
random forest	0.9257	Logistic Regression	0.8929
SVM linear	0.9221	bagging	0.8845
SVM Gaussian	0.9213	Naïve Bayes	0.8745

Fig. 2. Area under curve (AUC) calculation for Response Operating Characteristic curve.

Rather than the AUC measure, it is more meaningful in practice to compare an early portion of the ROC plots because an early portion shows among high-ranked alarms how many true and false alarms are mixed. Fig. 3 shows the early portion of the ROC plots for the eight classification methods. It shows that the random forest method is most effective in excluding false alarms from high-scored ones. It's plot is the nearest to the right angle. Later, however, boosting catches up with random forest: though it mixes more false alarms in high scores than random forest does, it later mixes less false alarms.

In numbers, the effectiveness of the top two methods (random forest and boosting) is as follows. Suppose the user sees higher-scored alarms earlier. Then, by the time 50% (950) of the 1900 true alarms[1] 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

[1] The total 1900 true alarms are from 100 test sets, each of which has 19 true and 81 false alarms.

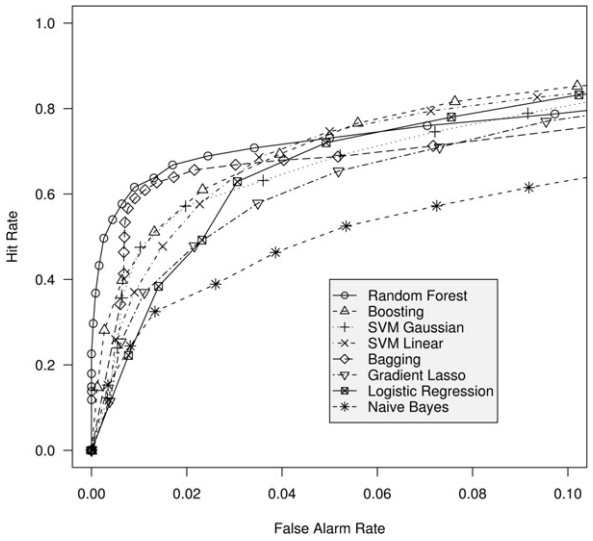

Fig. 3. ROC (Receiver Operating Characteristic) curves of the eight classification methods. X-axis: #false alarms/total #false alarms. Y-axis: #true alarms/total #true alarms.

alarms have been seen. Meanwhile, for boosting, 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 scores. For random forest, at the score threshold by which 50% (4050) of the 8100 false alarms are filtered out, 5.10% (97) of the 1900 true alarms are included; for boosting, this number is only 4.47% (86). The preference between boosting and random forest depends on the relative importance of missing true alarms, and of checking false alarms.

That random forest and boosting work better in our case agrees with other empirical comparison studies [2, 18,15,14]. Random forest and boosting are both ensemble methods (where a collection of decision trees vote on a decision) that have strategies for reducing, if any, a bias of decision trees. In our case too, reduced bias (i.e., an increased variety) in decision trees appears to improve the classification accuracy. Boosting directly reduces a bias by giving an extra attention to marginal data (mis-classified or near-to-the-classification-boundary data) while random forest indirectly dilute a bias by introducing extra randomness. In boosting, tree-growing data set (a bootstrap sample from the training set) is selected such that marginal data near to the classification boundary by the previous tree, which may be the victims of a bias, are fitted again in the current decision tree. Random forest grows each tree with an independent random bootstrap sample, and for an additional randomness, for each node of each tree only a random subset of the symptoms (attributes) are considered to find the best splitter.

Trained classifiers are not biased for a particular test population. Though this is expected because we trained the classifiers with a pool of multiple codebases (Linux and non-Linux codes), to see whether the trained classifiers perform well only on a particular test population or not, we measured the fraction of Linux alarms among the correctly classified alarms in tests. We measure the fraction for each method varying the threshold values from 0.1 to 0.8. The range of the fraction for all the eight classifiers is from 39.8% to 69.5%. That is, in correctly classified alarms, Linux and non-Linux alarms are about half and half, meaning that the trained classifiers are not biased for either Linux or non-Linux alarms.

Regarding the relative importance[2] (Fig. 4) of the 20 symptoms during the random forest method, 10 most effective symptoms include those from all three classes: all 3 result symptoms, 3 from 5 semantic symptoms, and 4 from 12 syntactic symptoms. This indicates that each of the three symptom classes is meaningful for classification. Among the top 10 symptoms, result symptoms are the most effective indicators (ranked 1st, 2nd, and 4th), followed by semantics symptoms (ranked 3rd, 5th, and 7th) and by syntactic symptoms (ranked 6th, and from 8th to 10th). Regarding the bottom 10 syntactic and semantic symptoms, they are not clearly comparable. While semantic symptom Join$N$ is next to the least effective, another semantic symptom InStructure is better than six other syntactic symptoms.

4. Conclusion

Random forest [4] and boosting [10] generated the most accurate classifications, closely followed by support vector machines [17], among the eight classification methods trialled. Our results may hint to static analysis designers, who use semantics-based framework such as abstract interpretation [6–8] to develop bug-finding analyzers, that ensemble methods are effective in ranking the output alarms, particularly if the methods are tuned, as in random forest [4] and boosting [10], to increase the varieties of decision trees. The three classes of symptoms (syntactic, semantics, and result symptoms) for signaling the analysis (im)precision are common across semantics-based static analyzers.

Although improving the accuracy of the static analysis is the orthodox approach to increasing the effectiveness of bug-finding static analyses, it appears that there

[2] The relative importance is measured by the Gini index [11].

Rank	Symptom	Class	Rank	Symptom	Class
1	FiniteIndex	result	11	InFunParam	syntactic
2	TopIndex	result	12	InNestedBranchBody$N$	syntactic
3	FailPrune	semantic	13	InStructure	semantic
4	HalfInfiniteIndex	result	14	InLoopCond	syntactic
5	Prune	semantic	15	CallFuncPtr	syntactic
6	CallFunc	syntactic	16	AfterReturn	syntactic
7	FailNarrow	semantic	17	InBranchCond	syntactic
8	AfterBranch	syntactic	18	InRightOfAnd	syntactic
9	AfterLoop	syntactic	19	Join$N$	semantic
10	InNestedLoopBody$N$	syntactic	20	InNestedFuncParam	syntactic

Fig. 4. Symptom's relative importance in random forest.

is value in applying statistical classification method. Classification methods are relatively easy and effective to improve the usability of bug-finding static analyses.

References

[1] The R Project for Statistical Computing, http://www.r-project.org.

[2] E. Bauer, R. Kohavi, An empirical comparison of voting classification algorithms: Bagging, boosting, and variants, Machine Learning 36 (1998) 105–139.

[3] L. Breiman, Bagging predictors, Machine Learning 24 (2) (1996) 123–140.

[4] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32.

[5] W.R. Bush, J.D. Pincus, D.J. Sielaff, A static analyzer for finding dynamic programming errors, Software—Practice and Experience 30 (2000) 775–802.

[6] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of ACM Symposium on Principles of Programming Languages, January 1977, pp. 238–252.

[7] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proceedings of ACM Symposium on Principles of Programming Languages, 1979, pp. 269–282.

[8] P. Cousot, R. Cousot, Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, in: PLILP'92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming, Springer-Verlag, Berlin, 1992, pp. 269–295.

[9] T. Fawcett, ROC graphs: Notes and practical considerations for data mining researchers, Technical Report HPL-2003-4, HP Labs, 2003.

[10] J.H. Friedman, Greedy function approximation: a gradient boosting machine, Annals of Statistics 29 (2001) 1189–1232.

[11] T. Hasite, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference and Prediction, Springer, Berlin, 2001.

[12] Y. Jung, J. Kim, J. Shin, K. Yi, Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis, in: SAS'05: 12th Annual International Static Analysis Symposium, in: Lecture Notes in Computer Science, vol. 3672, Springer, Berlin, 2005, pp. 203–217.

[13] T. Kremenek, D. Engler, Z-ranking: Using statistical analysis to counter the impact of static analysis approximations, in: R. Cousot (Ed.), SAS'03: Proceedings of the 10th Annual International Static Analysis Symposium, in: Lecture Notes in Computer Science, vol. 2694, Springer, Berlin, 2003, pp. 295–315.

[14] J.W. Lee, J.B. Lee, M. Park, S.H. Songa, An extensive comparison of recent classification tools applied to microarray data, Computational Statistics and Data Analysis 48 (2005) 869–885.

[15] R.A. McDonald, D.J. Hand, I.A. Eckley, An empirical comparison of three boosting algorithms on real data sets with artificial class noise, in: MCS 2003: 4th International Workshop on Multiple Classifier Systems, in: Lecture Notes in Computer Science, vol. 2709, Springer, Berlin, 2003, pp. 35–44.

[16] R. Tibshirani, Regression shrinkage and selection via the Lasso, Journal of the Royal Statistical Society, Ser. B 50 (1) (1996) 267–288.

[17] V.N. Vapnik, Statistical Learning Theory, John Wiley & Sons, Inc., 1998.

[18] B. Wu, T. Abbott, D. Fishman, W. McMurray, G. Mor, K. Stone, D. Ward, K. Williams, H. Zhao, Comparison of statistical methods for classification of ovarian cancer using mass spectrometry data, Bioinformatics 19 (September 2003) 1636–1643.