Buffer Overflow Detection for C Programs is Hard to Learn

Yang Zhao
Oracle Labs, Australia

Xingzhong Du*
Oracle Labs, Australia

Paddy Krishnan
Oracle Labs, Australia

Cristina Cifuentes
Oracle Labs, Australia

ABSTRACT

Machine learning has been used to detect bugs such as buffer overflow [6, 8]. Models get trained to report bugs at the function or file level, and reviewers of the results have to eyeball the code to determine whether there is a bug in that function or file, or not. Contrast this to static code analysers which report bugs at the statement level along with traces [3, 7], easing the effort required to review the reports.

Based on our experience with implementing scalable and precise bug finders in the Parfait tool [3], we experiment with machine learning to understand how close the techniques can get to a precise static code analyser. In this paper we summarise our finding in using ML techniques to find buffer overflow in programs written in C language. We treat bug detection as a classification problem. We use feature extraction and train a model to determine whether a buffer overflow has occurred or not at the function level. Training is done over labelled data used for regression testing of the Parfait tool. We evaluate the performance of different classifiers using the 10-fold cross-validation and the leave-one-out strategy. To understand the generalisability of the trained model, we use it on a collection of unlabelled real-world programs and manually check the reported warnings.

Our experiments show that, even though the models give good results over training data, they do not perform that well when faced with larger, unlabelled data. We conclude with open questions that need addressing before machine learning techniques can be used for buffer overflow detection.

ACM Reference Format:
Yang Zhao, Xingzhong Du*, Paddy Krishnan, and Cristina Cifuentes. 2018. Buffer Overflow Detection for C Programs is Hard to Learn. In *Proceedings of (ISSTA Companion/ECOOP Companion'18).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3236454.3236457

1 AIMS

Our aim is to identify machine learning techniques that can be used to *replace* hand-crafted static analysis tools. Our overall goal is to reduce the effort required to develop static analyses to detect well known defects. Therefore, our experiments are aimed to determine if machine learning techniques can be used for this endeavour. More specifically, we investigate *can machine learning techniques be used to find buffer overflow in C code, and how does it compare against hand-crafted static analysis tools like Parfait [3].*

We treat bug detection as a classification problem and attempt to solve it with supervised learning techniques. This work is an extension of our previous work [1] which used both manually identified and CNN-based features with other off-the-shelf learning algorithms.

We experiment with features that are extracted from a structural representation of the program and use existing classifiers to determine their effectiveness. For training we use standard micro benchmark suites like SAMATE, Iowa and Zitser from BegBunch [2] as they contain labelled defects. The dataset contains a few thousand buggy and non-buggy programs, with many more buggy programs than non-buggy. Here we focus on the traditional "feature extraction plus classifier" approach. Initially, we experiment with classification at at function level rather than file or statement level. This is because file-level classification is too coarse grained, and it is not clear if statement by themselves contain enough features for training purposes.

2 FEATURE EXTRACTION

Our first approach is to reuse the *code property graph* (CPG) [9], of a program to create a 45-dimensional feature vector for each function based using other reported results [8, 9] including number of local variables, function calls, array indexing and member accessing, cyclomatic complexity, syntax-only symbol sanitisation information on sampled variables, and taint information to a variable.

Our second approach is to automatically learn representative features from source code or its LLVM representation, which is also used by the Parfait tool, using unsupervised learning techniques. We evaluate the suitability of recurrent neural network (RNN) with LSTM that has been explored in the context of static analysis [5]. We trained character-level RNNs on both C source code and its LLVM assembly code separately using the entire Linux kernel codebase. The trained RNN's last hidden layer can be used to encode any given input. That is, we are using the RNN as the autoencoder to transform a program function input as either as source code (RNN-source) or LLVM intermediate code (RNN-LLVM) into its feature vector.

3 CLASSIFICATION AND VALIDATION

We use three classifiers, viz., naïve Bayesian classifier (NBC), support vector machine (SVM) and gradient boost tree (XGBoost), to evaluate the effectiveness of the features that have been identified.

*Xingzhong Du was an intern at Oracle Labs. His current affiliation is with The University of Queensland. He is reachable at xingzhong.du@uqconnect.edu.au.

ISSTA Companion/ECOOP Companion'18 ,
July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5939-9/18/07…$15.00
https://doi.org/10.1145/3236454.3236455

We use the harmonic mean of precision and recall, F1 score, to compare the accuracy of results. The F1 score is relevant given that the distribution of source code is often skewed to non-buggy programs. We wish to detect and avoid using classifiers that are biased towards indicating that the function has no bug.

Using CPG features, the XGBoost classifier exhibited the best behaviour with an F1 score of 82%. This contrasts with the scores of 67% and 80% for NBC and SVM respectively.

For the features identified by the RNN techniques we used only the XGBoost classifier in the evaluation. The results were very poor with F1 scores of 4% and 10% for RNN-source and RNN-LLVM respectively. While the reasons for this poor performance is not clear, we speculate that our current technique of using RNNs cannot go deep into the semantic level. Thus it is unable to learn about how data values are transferred in a program. RNNs specifically, and deep learning in general, exhibit a strong correlation between training data set, computational scale and model accuracy [4], and we believe our training data is still in the "small data region." So we discarded the RNN models from our other investigations.

The F1 score obtained by running our internal static tool Parfait on the same data set is 95% which suggests that the CPG+XGBoost (CPGX) approach can be used in practice.

4 GENERALISABILITY

As learning-based methods are often less effective when the test data have different distribution compared to training data, it is important for evaluate CPGX in different settings. We evaluate CPGX using the leave-one-out technique on unseen code – we select the three largest suites in our data set and treat two of them as training set, while the other as testing set. This leave-one-out strategy simulates the situation that the training data and testing data may have different distributions. It turns out that no matter how we arrange and divide the three benchmark suites of BegBunch, the F1 score for the leave-one-out technique is less than 10%. One can thus conclude that the features in the CPGX experiment are unstable and using CPGX in practice is risky.

In order to further understand the limitations of using CPGX on real code, we ran CPGX on the Calysto[1] benchmarks. While CPGX produced 111 reports, a manual inspection identified most of them as false positives. There were two main reasons for this. The first is that CPGX has been trained a function at a time. Thus it learns only about intraprocedural errors while the programs in the Calysto suite require an interprocedural analysis to remove the false positives. The second reason is that CPGX is unable to handle even slightly complex array operations and concludes that the chances of a bug being present when an array is used is high.

Our initial conclusion from these experiments is that structural code features that we have considered, on their own or coupled with shallow semantic features from the CPG implementation, are not enough to train a model that can detect buffer overflow errors. Although more extensive experimentation is required our believe is that structural features are not sufficient by themselves and more precise semantic features need to be considered.

[1] http://www.domagoj-babic.com/index.php/ResearchProjects/Calysto

5 CONCLUSIONS AND OPEN QUESTIONS

Based on our experiments with the available data we conclude that CPGX showed the most promise when using the 10-fold cross validation technique. However the learned model was not reliable as its performance using the leave-one-out and on unseen code was not useful. Using RNNs to automatically learn features was also not successful.

Some of the open challenges that remain include:

- What is the set of necessary features that can be used effectively for bug finding?
- What are the characteristics, such as size, distribution of buggy vs non-buggy, of the data set that will enable the use of deep learning techniques?
- Having identified the required characteristics, can we gather sufficient labelled data that satisfy them?
- What is an appropriate classification technique and what is its influence on the feature set?
- Are there off-the-shelf techniques that are sufficiently powerful for bug detection or are custom machine learning algorithms required? Can probabilistic reasoning techniques be used to guide the learning algorithms?

REFERENCES

[1] T. Chappell, C. Cifuentes, P. Krishnan, and S. Geva. Machine learning for finding bugs: An initial report. In *Proceedings of the IEEE MaLTeSQuE Workshop*, pages 21–26, 2017.
[2] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. BegBunch: Benchmarking for C bug detection tools. In *Proceedings of the 2009 International Workshop on Defects in Large Software Systems*, July 2009.
[3] C. Cifuentes, N. Keynes, L. Li, N. Hawes, and M. Valdiviezo. Transitioning Parfait into a development tool. *IEEE Security and Privacy*, 10(3):16–23, May/June 2012.
[4] J. Hestness, S. Narang, N. Ardalani, G. F. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou. Deep learning scaling is predictable, empirically. *CoRR*, abs/1712.00409, 2017.
[5] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *MAPL*, pages 35–42. ACM, 2017.
[6] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *NDSS*, 2018.
[7] LLVM/Clang Static Analyzer. http://clang.llvm.org/StaticAnalysis.html. Last accessed: 1 December 2010.
[8] B. M. Padmanabhuni and H. B. K. Tan. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *COMPSAC*, pages 450–459. IEEE, 2015.
[9] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *S&P*, pages 590–604. IEEE, 2014.