A Machine Learning-based Software Code Weakness Detect Approach

AiGuo Lu, KangYi Luo
Wuhan Digital Engineering Institute, Wuhan 430205, China
laigwx@163.com

Abstract. In recent years, machine learning and deep learning have yielded immense success on computer vision, speech recognition and natural language processing. However, exploration of machine learning on software code weakness detect haven’t been fully investigated. Software source code could be thought of a kind of natural language but has the intrinsic difference from common natural language. In this work we propose a novel machine learning based approach to tackle the problem of software code weakness detection. Differing from traditional algorithm based tools, flawfinder, for example, we do not design the code weakness detection processing logics directly, instead we propose to use a variant version of Attribute-Relation File Format (ARFF) to preprocess the source code and utilize a state of the art classification algorithm in machine learning to build a unite code weakness detection model to learn the processing logic indirectly as a general framework. In order to exam the detection performance of our model, we collected a couple of datasets for each common weakness of Common Weakness Enumeration (CWE) List. Extensive results on these collected datasets show that our model achieves comparative results from traditional algorithm based tools in both false positive rate and false negative rate.

1. Introduction
In recent years machine learning especially deep learning has become the center of attention as a subfield of artificial intelligence. Machine learning algorithms build a goal to learn patterns or knowledge from data so that they can make predictions for unseen or unknown future. Since machine learning is capable of building model to make predictions, computer scientists began to design intelligent software code weakness detection systems. These kind of systems can not only detect code weakness but also can learn from previous mistakes so as to improve its future detection performance.

Feature extraction of source code technique plays an important role in software weakness detection. [9] proposed a semi-automatic methodology of classification and the classification itself, which exposes relationships among software weaknesses, security principles and security pattern but is not focused on the foundational feature extraction stage. Most of previous work made use of sequential description method which represents code features as sequences of system calls. This kind of method describes code features as code execution process, but is limited to the order of code execution, thus it’s easy to be interfered by order free execution exchange. In recent years, researchers began to represent inner relationships of source code with dependency relationships and employ system calls and data dependency graph between system calls to describe features so as to dynamically mining differential subgraph between evil and normal code. However, they didn’t consider the effect of control dependency relationships and suffers from the choice of comparison samples and the continuity of differential subgraph.
In order to solve the feature extraction problem mentioned above, we propose to use a variant Attribute-Relation File Format method to represent each line of code and utilize lexical analysis to do generalization processing. Details could be found in sub sections. In the following section, we will firstly present our feature extraction method in detail and the machine learning algorithms we utilized in section 2. Experiment settings and test results will be in section 3. Finally, we conclude this paper and propose future work directions in section 4.

2. Algorithm

In this section we introduce our feature extraction algorithm for source code in detail and further introduce our employed machine learning algorithms. We will give an overview of our proposed detection algorithm and show how it works and give detailed algorithm explanations in further subsections.

Firstly, we preprocess C/C++ source code files with general lexical analysis which generates uniform and general symbol representing raw source code. Next we utilize a novel feature extraction method to mining useful feature information from data and make numerical representations which includes data type, function name, parameter info, variable scopes and the range of variables and finally incorporate all the information into a feature dict. For each category of CWE [8] code weakness, we train a couple of classification models from a machine learning algorithm library and choose one of the best ones as the final detection model.

As for code detection we employ the above preprocessing strategy to make numerical representation of source code and load trained models for each CWE [8] category with which we detect each category of CWE [8] weakness for the target code and gives the final detection results report.

2.1. Datasets generation

We adopt experiment datasets generated from static analysis tool from Central Authentication Service (CAS) of America National Security Agency (NSA). Each file of the dataset concludes error code samples and right ones. We separate the error codes and right ones into different files and remove code comments and unrelated contents as well.

2.2. Contrast feature extraction

We propose to use ARFF (Attribute-Relation File Format) [1] to transform each line of code as follows, for example:

“if (data! = NULL)” transformed to ARFF format as:

```xml
<keyword="if", par1="data", op1="! =", value="NULL"/>
```

From which we can see the properties used by ARFF: "keyword" represents keyword used in C/C++ source code, “function” represents function used in C/C++ and “par” represents for parameters for functions and “op” represents for operations of C/C++ and “value” as corresponding values for variables been used.

Next we utilize lexical analysis to make generalization processing for ARFF formatted info as:

```xml
<keyword="if", variable="data", par1="data", op1="! =", value="NULL"/>
```

Some self-defined elementary type described in Table 1.

| Type Name | Type Description |
|-----------|------------------|
| variable  | Match any variable name |
| parameter | Match any parameter  |
| Str       | Match any string   |
After transformation of raw data files, since right codes are corrected based on the error ones, we compare the converted version of error code and right one to obtain the difference between the two files which concludes error code sentences and corrected ones with which we can track the execution process and capture the states of all the variables and after store the difference we reach our final feature dict. The whole feature extraction process is presented in Algorithm 1.

2.3. Classification algorithm selection
There exists a number of classification algorithms in machine learning and each has its own advantages and disadvantages.

A Support Vector Machine (SVM) [5] is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data, the algorithm outputs an optimal separating hyperplane which categorizes new examples. Another classification algorithm called logistic regression (LR) [6, 7] uses a logistic function to model a binary dependent variable which is very popular in industry due to its easy to use, easy to scale and parallelizable ability. Gradient boosting Decision Tree (GBDT) [3, 4] is a machine learning algorithm for regression and classification problems, which produces a prediction model in the form of an ensemble of decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

We built a list of these models for each of the CWE category dataset since each category has its own distinct features. We choose an optimal one for each category as a final detection model.

### Algorithm 1 Contrast feature extraction process

| func_name | Match any function name |
|-----------|-------------------------|
| class_name | Match any class name |
| ops       | Match any operator |
| num       | Match any number |

Input: The set of CWE weakness $C_1$; The set of bad files with specified CWE weakness $B_1$; The set of good files without CWE vulnerabilities $G_1$;
Output: Feature dictionary for specified CWE defect.

1: for each $c \in C_1$ do
2:     initialize a set $B_2$ of bad files from $B_1$ with only $c$ weakness;
3:     for $b \in B_2$ do
4:         initialize a set $G_2$ of good files from $G_1$ and correspond to file $b$;
5:         convert file $b$ to ARFF format;
6:         traverse ARFF file and save to bad dictionary bad_dict;
7:     for $g \in G_2$ do
8:         convert file $g$ to ARFF format;
9:         for each item in ARFF formta do
10:             if item in bad_dict do
11:                 save item into good dictionary good_dict;
12:             end if
13:         end for
14:     end for
15:     combine bad_dict and good_dict into feature dictionary of weakness $c$;
16: end for
3. Experiment and result

In this section, we provide the experiment setting and test results on our collected datasets.

Our experiment environment is common personal computer and we do not use GPU since we only use machine learning tools. We adopt experiment datasets generated from static analysis tool from CAS of America National Security Agency. We use scikit-learn [7] as the machine learning framework.

The dataset information are summarized in Table 2. Each dataset named with CWE-number for which number means the distinct CWE category for their intrinsic similarity. Positive samples means that these are the good sample which does not have the corresponding CWE weakness and negative samples means these samples are of the opposite. We split these datasets by the ratio of 0.8 and 0.2 for train and test respectively.

| CWE121 | CWE122 | CWE190 | CWE191 | CWE416 | CWE606 |
|--------|--------|--------|--------|--------|--------|
| Positive samples | 310 | 352 | 664 | 638 | 279 | 212 |
| Negative samples | 126 | 117 | 140 | 122 | 81 | 65 |

We present experiment test results for logistic regression (LR) and gradient boosting decision tree (GBDT) and removed results for SVM since it’s not appropriate for these problems. We set LR with the l2 penalty and “class_weight” as 0.3 since the data is imbalance and tuned hyper parameters for GBDT with “n_estimators” and learning rate. We use false positive rate (FPR) and false negative rate (FNR) as experiment metrics. We also tested the datasets with the tool flawfinder[2] and find that this kind of tools has a very low false negative rate with almost 0 percent and very high false positive rate since it report every function that has been used. Our experiment results are presented in Table 3 and Table 4.

| Algorithm | CWE121 | CWE122 | CWE190 | CWE191 | CWE416 | CWE606 |
|-----------|--------|--------|--------|--------|--------|--------|
| LR        | 0.30   | 0.11   | 0.16   | 0.05   | 0.12   | 0.01   |
| GBDT      | 0.08   | 0.23   | 0.18   | 0.16   | 0.22   | 0.04   |

| Algorithm | CWE121 | CWE122 | CWE190 | CWE191 | CWE416 | CWE606 |
|-----------|--------|--------|--------|--------|--------|--------|
| LR        | 0.16   | 0.08   | 0.20   | 0.18   | 0.08   | 0.05   |
| GBDT      | 0.02   | 0.17   | 0.33   | 0.30   | 0.40   | 0.18   |

From the test results we can see that our models have much better performance than tools like flawfinder on false positive rate as well as false negative rate which are very important metrics for evaluating detection performance and we can also find that different classification algorithm has different performance on each of the CWE category. GBDT has much better performance that LR for CWE121 and LR exhibits his dominant position for the rest of CWE category.
4. Conclusion and future work
In this work, we devised a general machine learning based software weakness detection model. Our model framework is simple and generic; however, it is not limited to algorithms and models presented in this work, but is designed as a guideline for developing machine learning or deep learning based models for software code weakness detection. This work opens up a new avenue of research probabilities for software code weakness based on machine learning.

We haven’t investigated deep learning algorithms in this model framework, in future work we may take advantage of the powerful capability of deep neural networks to build a more powerful and generic model.

References
[1] Tribus H. Static Code Features for a Machine Learning based Inspection: An approach for C[J]. 2010.
[2] flawfinder : https://dwheeler.com/flawfinder/
[3] Ayyadevara V K. Gradient Boosting Machine[J]. 2018, 10.1007/978-1-4842-3564-5(Chapter 6):117-134.
[4] Friedman J H. Greedy function approximation: A gradient boosting machine.[J]. Annals of Statistics, 2001, 29(5):1189-1232.
[5] Bloodgood M. Support Vector Machine Active Learning Algorithms with Query-by-Committee Versus Closest-to-Hyperplane Selection[J]. 2018.
[6] Yang Y, Loog M. A benchmark and comparison of active learning for logistic regression[J]. Pattern Recognition, 2018, 83(C):401-415.

[7] Pedregosa F, Gramfort A, Michel V, et al. Scikit-learn: Machine Learning in Python[J]. Journal of Machine Learning Research, 2013, 12(10):2825-2830.

[8] CWE : http://cwe.mitre.org/

[9] Regainia L, Salva S, Ecuhcurs C. A classification methodology for security patterns to help fix software weaknesses[C]/ Computer Systems & Applications. 2017.