POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects

Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan
School of Information Technology, Deakin University, Australia
Geelong, VIC
{lingu,jun.zhang,wei.luo,l.pan}@deakin.edu.au

Yang Xiang
Digital Research & Innovation Capability Platform, Swinburne University of Technology, Australia
Melbourne, VIC
yxiang@swin.edu.au

ABSTRACT

In cybersecurity, vulnerability discovery in source code is a fundamental problem. To automate vulnerability discovery, Machine learning (ML) based techniques has attracted tremendous attention. However, existing ML-based techniques focus on the component or file level detection, and thus considerable human effort is still required to pinpoint the vulnerable code fragments. Using source code files also limit the generalisability of the ML models across projects. To address such challenges, this paper targets at the function-level vulnerability discovery in the cross-project scenario. A function representation learning method is proposed to obtain the high-level and generalizable function representations from the abstract syntax tree (AST). First, the serialized ASTs are used to learn project independence features. Then, a customized bi-directional LSTM neural network is devised to learn the sequential AST representations from the large number of raw features. The new function-level representation demonstrated promising performance gain, using a unique dataset where we manually labeled 6000+ functions from three open-source projects. The results confirm that the huge potential of the new AST-based function representation learning.

KEYWORDS

vulnerability detection; cross-project; AST; representation learning

1 INTRODUCTION

Vulnerability detection is an important problem for mitigating security risks in software. Early and accurate detection is the key that makes machine learning (ML) based automatic vulnerability detection techniques a preferred approach, as manual inspection has been infeasible with program source codes growing exponentially. Scandariato et al. [5] applied ML techniques for detecting vulnerable components on Android applications. Shin et al. [6] used models trained on early versions of Firefox and Linux kernel for vulnerable files detection on subsequent versions. These approaches mainly focus on component- or file-level vulnerability detection. A finer-grained approach was presented by Yamaguchi et al. [7] for extrapolating vulnerable functions. Nevertheless, their method works on the within-project domain.

CCS'17, , Oct. 30–Nov. 3, 2017, Dallas, TX, USA.

ACM ISBN ISBN 978-1-4503-4946-8/17/10.
https://doi.org/http://dx.doi.org/10.1145/3133956.3138840

In this paper, we propose an approach for function-level vulnerability detection on cross-project scenario. We overcome the difficulty of obtaining manual labels by leveraging well-understood complexity metrics (used as a *proxy*), which can be automatically generated at large scales. Such complexity metrics data are subsequently used to bootstrap the generation of rich representations of the abstract syntax trees (ASTs) of functions. Our approach builds on the assumption that vulnerable programming patterns are associated with many potential vulnerabilities, and these patterns can be discovered by analyzing the program's ASTs. To capture local and relational features in a function, we use bi-directional Long Short-Term Memory (LSTM) [3] networks for learning high-level representations of ASTs. Our empirical studies illustrate that the obtained representations reveals important signals which can distinguish between neutral and vulnerable functions.

Our contributions can be summarized as follows:

- We propose a learning framework for functions-level vulnerability detection on cross-project domains.
- We develop an approach to extract the sequential features of ASTs which represent the structural information of functions for vulnerability detection.
- We implement a stacked LSTM network and use a proxy as the substitute of data labels for acquiring high-level representations of ASTs which can be used as indicators for vulnerability detection.

2 FUNCTION REPRESENTATION LEARNING

Figure 1 illustrates the work flow of our proposed framework which contains 4 stages. The first stage is data collection. Our experiments include three open-source projects: LibTIFF, LibPNG and FFmpeg whose source code can be downloaded from Github. We extract raw features from the ASTs of functions in the second stage. In stage three, we design a stacked LSTM network and introduce a proxy for learning AST representations of functions. The last stage is to examine prediction capability of the learned representations in detecting vulnerable functions on real-world cases.

2.1 AST Sequential Processing

We used "*CodeSensor*[1]", which is a robust parser implemented by [8] based on the concept of *island grammars* [4], so we could extract ASTs from source code without a working build environment.

An AST represents programming patterns by depicting the structural information of the code (i.e. a function). It reveals the relationships of components of a function in a tree view and contains

[1] http://codeexploration.blogspot.com.au/

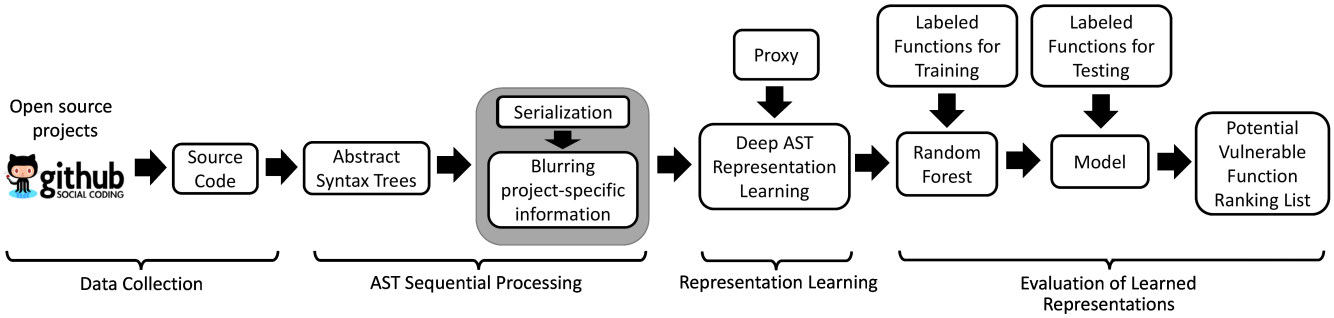

Figure 1: The work flow of our proposed framework

control flow at function level. To preserve the structural information, we use depth-first traversal to map the nodes of ASTs to elements of vectors so that each node will become an element of a vector. The sequence of elements in a vector partially reflects the hierarchical position of the nodes in an AST. After mapping the AST of a function to a vector, it will be in a form like: [*foo, int, params, int, x, stmnts, decl, int, y, op, =, call, bar,...*]. The first element *foo* is the name of the function, and the second element *int* denotes the return type of function *foo*. The third and fourth elements *params* and *int* specify that function *foo* takes a parameter which is an *int* type, and so on. For this textual vector, we treat it as a "sentence" with semantic meanings. The semantic meaning is formed by the elements of the vector and their sequence.

Due to our data being collected from three different software projects which follow distinct naming conventions, a universal naming criterion is required to normalize the project-specific names. For instance, it is very unlikely for project LibPNG to have a function named exactly the same in FFmpeg. What we do is to blur the names that are project-specific and replace them with "proj-specific" terms.

With function-level vulnerability detection, a sufficient amount of labeled data are required to train a statistical model. However, such labels are usually obtained through manual process. Instead of using actual labels, we use a proxy to approximate the functionality of the label. In this paper, we choose the proxy from code metrics as these metrics are quality measures for quantifying programs' complexity in software tests [2], therefore they were important indicators of faults and vulnerabilities in software [1] and [6]. Thanks to *Understand*[2], a commercial code enhancement tool, we are able to collect function-level code metrics from source code.

2.2 Deep AST Representation

There is a strong resemblance between ASTs and "sentences" in natural languages, which motivates us to apply LSTM for learning high-level representations. The vectors mapped from an AST containing components in a sequence reflect the tree structural information. Namely, the structure of an AST is partially preserved by elements assembled in a sequence through depth-first traversal. In this sense, the converted vectors contain semantic meanings which are formed by the sequential context of elements like [*main, int, decl, int, op,..., return*]. Altering the sequence of any element changes the semantic meanings. More importantly, there are connections among elements, which forms the context information. For instance, the first element of a vector is the name of a function, followed closely by its return type. The element immediately after function name "*main*" should be its return type such as "*int*" or "*void*", and should not be other C/C++ keywords such as "*decl*" or "*return*". LSTM' architecture is built for handling such data. So, we assume that a function with vulnerability will display certain "linguistic" oddities discoverable by LSTM.

Another reason which justifies the application of LSTM is that the vulnerable programming patterns in a function can usually be associated with more than one lines of code. When we map functions to vectors, patterns linked to vulnerabilities are related to multiple elements of the vector. The standard RNNs are able to handle short-term dependencies such as the element "*main*" which should be followed by type name "*int*" or "*void*", but they have problem of dealing with long-term dependencies such as capturing the vulnerable programming patterns that are related to many continuous or intermittent elements. Therefore, we use RNN with LSTM cells to capture long-term dependencies for learning high-level representations of potential vulnerabilities.

The architecture of our LSTM-based network contains 5 layers. The first layer is for embedding which maps each element of the sequence to a vector of fixed dimensionality. The second and third layers are stacked LSTM layers each of which contains 64 LSTM units in a bi-directional form (altogether 128 LSTM units per layer). The stacked LSTM layers are capable of learning higher-level abstractions. The last two layers are dense layers with "ReLU" and "Linear" activation functions for converging a 128-dimensionality output to a single dimensionality. The choice of loss function in our experiments is "MSE". We feed the network with extracted ASTs of the three open-source projects. Instead of using labels of the input ASTs, we used a code metric as a proxy for obtaining the representations which are the output of the third layer.

3 EMPIRICAL STUDY

We apply the random forest algorithm to evaluate the effectiveness of the learned AST representations.

Among the functions that we used for obtaining AST representations, we manually label them as vulnerable or neutral so we can examine how effective the learned representations are when using them as features for vulnerability detection. For acquiring

[2]https://scitools.com/

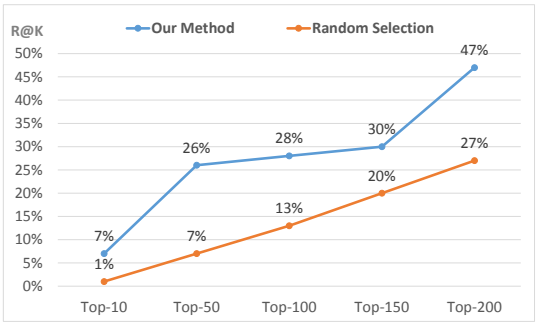

Figure 2: The R@K comparison between our method and random selection on cross-project scenario

vulnerable functions, we refer to the National Vulnerability Database (NVD)[3] and the Common Vulnerability and Exposures (CVE)[4], which are publicly available vulnerability data repositories. Our labels are according to the records until 20th July 2017. Then, excluding the identified vulnerable functions, we select the remaining functions as neutral ones. Before mapping ASTs to vectors, we truncate the trees that are more than 14 layers. When converting ASTs to vectors, we truncate the vectors containing more than 620 elements and pad them with 0s for vectors having fewer elements.

Due to the vulnerability inspection mainly being a manual process, it is not cost-effective for a practitioner to check on every function on the code base. It is practical to examine a small portion of code that is most likely to be vulnerable. For the evaluation, we retrieve a list of functions ranked by their probabilities of being vulnerable. The performance of our method is measured by the proportion of vulnerable functions returned in a function list. Hence, the metrics that we apply for evaluation is top-k recall ($R@K$). In our context, the R@K refers to the proportion of vulnerable functions which are in the retrieved top-k function list.

To evaluate the effectiveness of the learned representations on cross-project scenario, firstly, we combine the ASTs of the functions from three projects and feed them to the LSTM network. In this paper, we choose the "*essential complexity*" as the proxy for acquiring the representations. Secondly, we group the learned representations according to projects and use the FFmpeg and LibTIFF for training a random forest algorithm to test on the LibPNG. Totally, our training set contains 5736 functions among which there are 274 vulnerable functions. There are 750 functions in the testing set containing 43 vulnerable ones. Figure 2 illustrates that with our method, 20 vulnerable functions can be identified by checking the returned 200 functions. With random selection, one can only cover 27% of vulnerable functions by chance when examining 200 randomly selected functions.

To test that the LSTM networks truly pick up the vulnerable patterns hidden in the ASTs, we train a random forest classifier using essential complexity alone and compared whether its predicting power is stronger than the learned representations. The outcome shows that within top 50 retrieved functions, the learned representations exhibit more predicting power than the random selection scheme.

[3]https://nvd.nist.gov/
[4]https://cve.mitre.org/

4 DISCUSSION

This work, although in its early stage, has already demonstrated the feasibility and potential of learning deep representation from AST and proxy metrics. In particular, the project-independent AST representations provide new angles for vulnerable function discovery. We see several lines of research to extend the current work.

Alternative proxies. The proxy selected will affect the representation learned. We will evaluate other proxies, such as the line-of-code or cyclomatic complexity. Other metrics or information of programs can also serve as proxies. Moreover, to avoid overfitting, we can consider multiple proxies for guiding the new representation learning. This can be achieved using the multi-task deep-learning approach with shared modeling component.

The generalisability of representation can be further improved with a more flexible architecture such as the sequence-to-sequence network. This will eliminate the dependence on proxies. The current work benefits from the evaluation on a small set of labels manually curated. We are continuing the data labeling process. More labeled data will no doubt allow us to gain more insights into how deep learning can help reveal program vulnerabilities.

5 CONCLUSIONS

We propose an approach for automatic learning high-level representations of functions based on their ASTs. With an AST parser, we are able to extract functions' ASTs and convert them to vectors. To handle cross-project detection, we blur the project-specific contents while preserving the structural information by leveraging a stacked LSTM network for capturing representations that depict the intrinsic patterns of the vulnerable functions. Our experiment demonstrated that the learned representations were effective for cross-project vulnerability detection at function level.

ACKNOWLEDGEMENT

Guanjun Lin is supported by the Australian Government Research Training Program Scholarship. And this work was supported by the National Natural Science Foundation of China (No. 61401371).

REFERENCES

[1] Istehad Chowdhury and Mohammad Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture* 57, 3 (2011), 294–313.
[2] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 171–180.
[3] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[4] Leon Moonen. 2001. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 13–22.
[5] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.
[6] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. 2011. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787.
[7] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*. USENIX Association, 13–13.
[8] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 359–368.