Towards a Block-Level ML-Based Python Vulnerability Detection Tool

Amirreza Bagheri and Péter Hegedűs

Abstract: Computer software is driving our everyday life, therefore their security is pivotal. Unfortunately, security flaws are common in software systems, which can result in a variety of serious repercussions, including data loss, secret information disclosure, manipulation, or system failure. Although techniques for detecting vulnerable code exist, the improvement of their accuracy and effectiveness to a practically applicable level remains a challenge. Many existing methods require a substantial amount of human experts labor to develop attributes that indicate vulnerabilities. In a previous work, we have shown that machine learning is suitable for solving the issue automatically by learning features from a vast collection of real-world code and predicting vulnerable code locations. Applying a BERT-based code embedding, LSTM models with best hyperparameters were able to identify seven different security flaws in Python source code with high precision (average of 91%) and recall (average of 83%). Upon the encouraging first empirical results, we go beyond in this paper and discuss the challenges of applying these models in practice and outlining a method that solves these issues. Our goal is to develop a hands-on tool for developers that they can use to pinpoint potentially vulnerable spots in their code.

Introduction

Security bugs (i.e., vulnerabilities) in software are becoming more and more difficult to identify in today's applications, allowing hackers and attackers to profit from their exploit. Every year, tens of thousands of such vulnerabilities are discovered and fixed. Manually auditing source code and finding vulnerabilities is costly at best, if not impossible at all. In our previous work [1], we have shown that machine learning is suitable for solving the issue automatically by learning features from a vast collection of real-world code and predicting vulnerable code locations. The dataset was gathered from Github and contains Python code with a variety of vulnerabilities that is embedded into a vector space using one of three embedding models (word2vec, fasttext, BERT). Individual code tokens and their context are extracted from the source code of the vulnerable files to provide data samples for fine-grained analysis. We then trained various machine learning (ML) models to evaluate their effectiveness of identifying vulnerable code parts.

The entire training process may be divided into two parts: first, an embedding model is trained using its own parameters, such as min-count (how often a token has to appear in the training corpus in order to actually get assigned a vector representation) or iterations; and second, the system is trained using its own parameters, such as min-count or iterations. The code blocks can only be encoded in their vector representations after that. We found that LSTM models were the most suitable, thus we used them and trained with different hyperparameters, such as the number of neurons or dropout, in the second stage. Applying a BERT-based code embedding, LSTM models performed the best, they were able to identify seven different security flaws in Python source code with high precision (average of 91%) and recall (average of 83%).

Upon a successful empirical evaluation naturally comes the need of adopting the results in practice. However, there are several challenges of applying the above describe method in practice, offering as a developer tool. The training data samples are code snippets (extracted from vulnerability fixing commits) but when we apply vulnerability identification in practice, our input is the whole program. We need a method to locate code blocks within the program in an efficient way to be able to apply the code embedding and the model prediction. Furthermore,

these code blocks might overlap, so we need a way of aggregating block-level predictions. In this paper, we focus on solving these challenges and outline a possible developer tool that can be used by developers. We apply a small focus area and a sliding context window to divide the code into blocks. The focus area moves through the code, and with each successive step, the model gathers the surrounding information, generates a prediction based on that context as input, and uses that prediction as the focus area's vulnerability rating. The different confidence levels of the classification might be shown with distinct colors in a developer tool.

To sum up the contributions, we provide a block-level vulnerability prediction method practically applicable on Python code, as opposed to most other research initiatives, which are primarily focused on Java, C, C++, or PHP and do not provide guidance on practical application. Moreover, existing vulnerability prediction approaches are providing predictions at higher abstraction levels, like for methods, classes, or files, while we aim at a fine-grained smaller block-level prediction.

Related Works

In this section, we enumerate some of the most relevant related literature and highlight the main differences between those and the approach we take. Unlike Li et al. [2], Pang et al. [3], Hovsepyan et al. [7], and Dam et al. [4], we use a large code base and not just a few projects. The predictions are applicable not only within the same file or project, but to any other source code as well. A fine granularity is chosen in contrast to those approaches. All of the works listed classify entire files, or just take API and function calls into account in the case of Li et al. [2]. The work of Russell et al [5] and Ma et al. [8] are more analogous to our method, because vulnerabilities are detected not only at the file level, but also at specific locations inside the code, which is likely more beneficial for developers. The work of Yamaguchi et al. [6] does not convert source code into an abstract syntax tree-like structure, but instead treats it as plain text. It follows the natural premise and makes as few assumptions as possible, relying solely on the trained model to extract features from the source code. We utilized a long short term memory network, similar to those employed by Li et al. [2] and Dam et al. [4] but with a simpler architecture. Many alternative approaches use either different deep learning models or different machine learning approaches (support vector machines in the case of the contribution by Pang et al. [3]).

Architecture of Application

Our goal is to create a program that can anticipate the vulnerable areas of a software system unseen by the prediction model using our trained neural network that was specifically trained on vulnerable source code samples. To accomplish this, first we need to generate a model and test it with various hyperparameters in order to obtain the best possible output, which we already did and published in an earlier work [1]. Then, we apply the best model to the source code of an arbitrary program to predict its vulnerable areas. The machine learning algorithm we utilized for this was LSTM. After evaluating different settings of the hyperparameters of the LSTM, we used the best results acccording to the previous work, for the training part minimum ount was 10, while iteration count was 300, we used an architecture with 100 neurons, 100 epochs, and a dropout of 20% with an Adam optimizer. Following the LSTM layer, we used an activation layer, which is a dense output layer with a single neuron. The activation function used here was a sigmoid activation function, as the goal was to construct a prediction between 0 and 1 for the two groups of vulnerable and non-vulnerable code.

Since we trained the model on code blocks (coming from vulnerability fixing patches and the previous code state), applying the model in practice has a challenge of finding the right blocks within a whole project to which we can apply our trained model. We applied a small focus window that traverses through the whole source code in steps of length n. Some positions of

the focus window are depicted in blue in Figure 1. The focus window always starts and stops at a character that marks the end of a token in Python, for instance, a colon, a bracket or a whitespace to prevent breaking tokens.

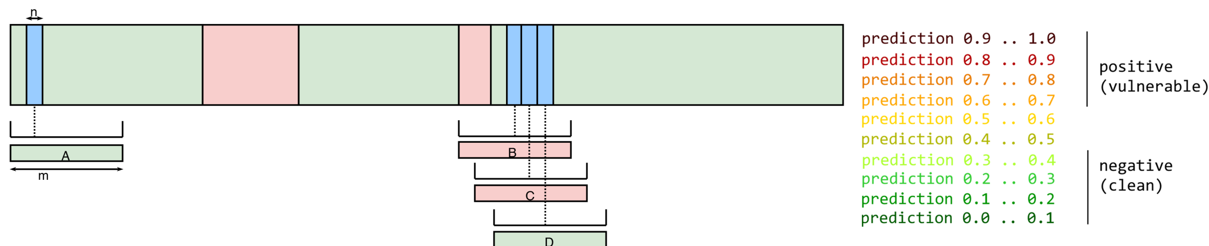

Figure 1: Focus window and its surrounding context

Figure 2: Confidence levels

For this focus window, the surrounding context of length m, also starting and stopping at the border of code tokens, is determined (where $m > n$). After evaluating different settings, we set the small window (n) to 5 tokens and the block (m) to 200 tokens. If the focus window is close to the beginning of the file, the context will mostly lie behind it (see Block A), and if it is located in the middle, the surrounding context will be spanning a snippet that lies equally before and after the focus window. This results in a number of overlapping blocks. If the whole block contains partially vulnerable code (as example blocks B and C), it is labeled as vulnerable, otherwise it is labeled as clean. This ensures that code snippets containing a vulnerability in them are marked as such. The parameters n and m are subject to optimization, their ideal values cab be determined empirically. As a result, each time the focus window shifts, more information about the block is obtained. As the focus region progresses through the code, the model absorbs all information about its surroundings, creates a prediction based on that context as input, and utilizes that prediction as the focus area's vulnerability rating. The different classification confidence levels as shown in Figure 2, could be highlighted with different colors. So if the block is fully clean, it will be green; if the block has overlaps with vulnerable snippets, the color will change to red depending on the amount of overlap there is; and if the block is completely vulnerable, it will be red. This visual aid can be very useful for the developer tool we are designing to implement.

```
    """
    sql_str = "SELECT id FROM wins_completed_wins_fy"
    if self.end_date:
        sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"

    with connection.cursor() as cursor:
        cursor.execute(sql_str)
        ids = cursor.fetchall()

    wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()

    for win in wins:
        yield self._get_win_data(win)

def get(self, request, format=None):
    end_str = request.GET.get("end", None)
    if end_str:
        try:
            self.end_date = models.DateField().to_python(end_str)
        except ValidationError:
            self.end_date = None
```

Figure 3: Vulnerability detection and its visualization

Figure 4: GitHub fix

Figure 3 and Figure 4 show a tiny example of a vulnerable code snippet with SQL injection and its fix on GitHub. We ran our prediction model on the vulnerable version of the code (i.e. before the fix), the results are shown in Figure 3. The command `cursor.execute` is used in the vulnerable code snippet to run a SQL query stored in the variable `sql_str`, which is constructed by concatenating strings together with other variables. The most vulnerable part is `cursor.execute(sql_str)`, which is 100 percent vulnerable after the focus window crosses it. Confidence levels of the whole block changes both the blocks before it and the blocks after it, this process just make confidece information by shifting focus window and in the end surrounding red blocks illustrate how close they are to that part.

Conclusion

This study presents a vulnerability detection method based on deep learning on source code. Its purpose is to relieve human vulnerability detection experts of the time-consuming and subjective effort of manually defining vulnerability detection criteria. Via LSTM models, this research demonstrates the feasibility of learning vulnerability attributes straight from source code using machine learning. It can detect seven different types of errors in Python source code. We were able to identify specific sections of code that are likely to be vulnerable, as well as provide confidence levels for our predictions. We get an accuracy of 93.8%, a recall of 83.2%, a precision of 91.4%, and an F1 score of 87.1% on average, the tool's output results are the 10 percent shuffled cross-validation result of the model evaluation based on our previous work[1], specific performance on subject systems yet to be performed. We also demonstrate how the model can be applied in practice, therefore it opens up the possibility of building a hands-on developer tool for detecting vulnerable code blocks in arbitrary Python programs. Moreover, the presented method is language agnostic, it can be adapted to other languages as well.

Acknowledgements

The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004) and supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (MILAB). Furthermore, Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences and the ÚNKP-21-5-SZTE-570 New National Excellence Program of the Ministry for Innovation and Technology.

References

[1] Bagheri, A. and Hegedűs, P., 2021, September. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In International Conference on the Quality of Information and Communications Technology (pp. 267-281). Springer, Cham.

[2] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681, 2018.

[3] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), pages 543-548, 2015.

[4] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for vulnerability prediction. arXiv preprint arXiv:1708.02368, 2017.

[5] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pages 757-762, 2018.

[6] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In Proceedings of the 28th Annual Computer Security Applications Conference, pages 359-368, 2012.

[7] IBM News Room. Ibm study: More than half of organizations with cybersecurity incident response plans fail to test them: Yet use of automation improved detection and containment of cyberattacks by nearly 25.cited 27.9.2019.

[8] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In European Symposium on Research in Computer Security, pages 229-246, 2017.