A transformer-based IDE plugin for vulnerability detection

Cláudia Mamede
FEUP, U.Porto
Porto, Portugal

Eduard Pinconschi
FEUP, U. Porto
Porto, Portugal

Rui Abreu
INESC-ID & FEUP, U. Porto
Porto, Portugal

ABSTRACT

Automatic vulnerability detection is of paramount importance to promote the security of an application and should be exercised at the earliest stages within the software development life cycle (SDLC) to reduce the risk of exposure. Despite the advancements with state-of-the-art deep learning techniques in software vulnerability detection, the development environments are not yet leveraging their performance. In this work, we integrate the Transformers architecture, one of the main highlights of advances in deep learning for Natural Language Processing, within a developer-friendly tool for code security. We introduce VDet for Java, a transformer-based VS Code extension that enables one to discover vulnerabilities in Java files. Our preliminary model evaluation presents an accuracy of 98.9% for multi-label classification and can detect up to 21 vulnerability types. The demonstration of our tool can be found at https://youtu.be/OjiUBQ6TdqE, and source code and datasets are available at https://github.com/TQRG/VDET-for-Java.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Computing methodologies → Machine learning**.

KEYWORDS

Vulnerability detection, Transformer, Plugin

ACM Reference Format:
Cláudia Mamede, Eduard Pinconschi, and Rui Abreu. 2022. A transformer-based IDE plugin for vulnerability detection. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3551349.3559534

1 INTRODUCTION

With the constant digitalization of our society, software usage is increasing daily. Similarly, vulnerabilities, defined as flaws that attackers can exploit [14], are also growing. Besides their cost and bad reputation, these tend to have a long vulnerability life cycle [1] and a disclosure time window of 312 days on average [2]. It is in the companies' best interests to ensure their software is free of flaws as soon as possible to reduce the risk of attack. Consequently, companies follow the shift-left principle [7] where a task traditionally done at a later stage of the process moves to an earlier phase of SDLC. Usually, security only concerns security auditors, who have the expertise to configure and use security tools, such as static and dynamic code analyzers, at a later stage of a project on deployed software [17]. This approach, along with the traditional techniques, does not suit the shift left principle due to the required expertise and considerable gap between software development and security.

Recent advancements with Deep learning (DL) in vulnerability detection [3] are appealing for merging security within the development as it eliminates the need for expert knowledge to configure and execute security tools. By changing the paradigm from rule-based program analysis tools to lightweight and efficient learning-based scanners integrated into development environments, developers can focus on quality from the start rather than waiting for errors to be discovered late in the SDLC. Hence, considering the progress with state-of-the-art DL techniques, namely the transformer model [18], and the lack of developer-friendly tools in this area, we introduce a proof-of-concept for vulnerability detection within the developer workflow to promote the early finding of software flaws. To this end, we propose a transformer-based VS Code extension to discover vulnerabilities in Java files.

Our contributions. This paper introduces a proof-of-concept for the aforesaid tool. We highlight the following three contributions:

- a **transformer-based VS Code extension** capable of identifying up to 21 CWEs and potential vulnerable code in Java projects, dubbed VDet for Java
- a **multi-label classification model** for vulnerability detection, with an accuracy of 98,9%.
- a **fine-grained dataset** with vulnerable and non-vulnerable Java methods and one-hot encoded labels for vulnerability detection.

Paper organization. This paper is structured as follows: Section 2 presents the background and related works on software vulnerability detection using transformers and other deep learning architectures. Section 3 introduces VDet for Java, outlining the main phases of its development. Section 4 concludes the paper, highlighting challenges and future steps.

2 BACKGROUND

The 2022 Open Source Security and Risk Analysis Report [1] identifies a slight decrease in the number of vulnerabilities found in projects, demonstrating that companies are investing in addressing software flaws. Although this process is usually reactive and organizations tend to wait for actual exploits to patch their software, there is a tendency towards a more proactive strategy. Developers increasingly follow the shift-left principle, focusing on quality from the start rather than waiting for late discovery of errors [21].

[1]https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559534

Hence, recent work explores DL-based systems for vulnerability detection to reduce the hurdles of analysis for developers and security experts associated with traditional techniques and anticipate the detection as much as possible. VulDeePecker [13], SySeVR [12] and μVulDeePecker [25] are examples of that. They rely mostly on Recurrent Neural Networks and (Bidirectional) Long Short-Term Memory neural networks. Despite having made significant progress in automation and accuracy, some problems still hinder these approaches. Managing longer sequences effectively, defining a suitable code entity that abstracts data without compromising relevant code characteristics, and the vocabulary explosion problem are some challenges of current solutions [11]. The transformer is a recent model that helps address these issues, and it has already been used for vulnerability detection, achieving state-of-the-art results, and proving the potential of the architecture [24]. It has a sequence-to-sequence architecture with an encoder-decoder structure and attention mechanisms. They rely entirely on self-attention to establish dependencies between inputs and outputs, allowing much of the computation to be performed in parallel.

Available tools: Few of these DL systems have found adoption by the software industry, with most tools still rule-based systems: Xie *et al.* [23] developed a plugin for Eclipse and Java that addresses common web application vulnerabilities dubbed ASIDE, and it also reminds programmers of safe programming practices within IDEs. Whitney *et al.* [20] upgraded ASIDE to promote a learning opportunity in the context of writing code. White *et al.* [19] designed an extension for Eclipse that identifies the violation of CERT rules in Java. Smith *et al.* [15] presented an innovative tool that not only detects vulnerabilities but also helps developers to solve them. Most of the proposed security solutions for developers are too confusing to use due to common issues. These include poorly presented outputs, a lack of actionable solutions for detected vulnerabilities, and too much noise between the alerts and what is relevant [9, 15].

3 VDET FOR JAVA

We coin our proof-of-concept as VDet for Java, a VS Code extension for vulnerability detection in Java source code, based on the transformer architecture. We focus on the Java programming language, as it is among the most investigated [4]. Our proof-of-concept follows the typical pipeline for vulnerability detection, including data gathering, pre-processing, learning, and evaluation [16]. For data-gathering, we collect, filter, and label the data. In the pre-processing, we transform the dataset into a suitable format for the model by tokenizing and encoding each code entity. Then, we fine-tune and evaluate JavaBert [5], a BERT-based model that uses only the encoder stack of the transformer for multi-label classification. Finally, we load the model into a server and establish communication with the extension to obtain the vulnerability identifications in code samples. This process is summarized in Figure 1.

3.1 Dataset: data extraction and preprocessing

As far as we know, there are no standardized datasets for machine learning-based vulnerability detection for the Java programming language. We build our dataset from one of the few available resources, the Juliet Test Suite for Java [2] from the NIST Software Assurance Reference Dataset Project (SARD). The Test Suite contains 28,881 files written in Java with examples of both vulnerable and non-vulnerable code. Each file is a test case labeled with a particular CWE tag corresponding to the security vulnerability of the associated program code and contains one or more methods.

[2]https://samate.nist.gov/SRD/testsuite.php

Table 1: Overview of our dataset, through the different filtering phases.

	Original	After NSDR	After LSR
# of samples	145 672	134 645	115 600
# non vulnerable samples	99 542	92 747	92 445
# vulnerable samples	46 130	41 898	23 155
# CWEs	111	21	21

Granularity: When training on program code, granularity usually ranges from statement level (i.e. fine granularity) to file level (i.e. coarse granularity) [16]. In a coarser granularity, there is more information that can be irrelevant and incorrectly represent the vulnerability. And would eventually require the need for expert knowledge to pinpoint the vulnerable code [10]. On the other hand, a finer granularity can filter out unnecessary information, promoting a focused detection and possible localization of the flaw [10, 13]. We use method-level granularity, as it comprises enough information to identify all CWEs present in the original Test Suite and still allows pinpointing vulnerabilities.

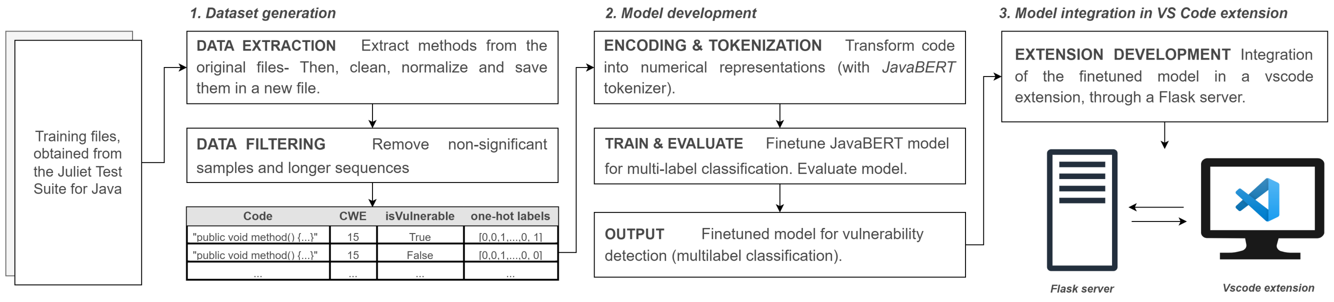
We constructed a parser that extracts the methods from the original files, normalizes them, and stores each of them into a new file. Words with “good” and “bad” prefixes/suffixes are removed from the original files so that the deep learning model would not rely on this to classify code. Similarly, the original method name is replaced with an abstract name (e.g., "method"), and comments are erased because they do not introduce vulnerabilities. Table 1 illustrates the evolution of the data set in size during the filtering phases explained below.

Non-significant data removal (NSDR): The distribution of samples per CWE in our dataset is not balanced. We remove CWEs that do not have enough samples for training to avoid compromising the learning process. For this, we calculate the mean samples per CWEs (i.e. *threshold = 1312*), defining it as the minimum number of required samples and filtering the dataset. Only 21 CWEs [3] are kept, reducing the total number of samples for training to 134,645.

Longer sequences removal for training purposes (LSR): BERT architectures consume up to 512 numerical tokens [6], so longer sequences should be encoded and either truncated or split into chunks before being fed to the model. The latter option is not appropriate when training a model for vulnerability identification. Since each chunk is treated independently, dividing the code into sections and assigning the same label to all could severely harm the model's accuracy. The complete code sequence is vital for the classification and cannot be fully understood if a chunk is missing. Therefore, we removed code sequences that exceeded 512 tokens, ending with 115,600 samples.

Finally, we one-hot encode the labels (the CWEs and “Vulnerable”). Li *et al.* compared a numeric encoding and a one-hot encoding

[3]CWEs: 113, 129, 134, 15, 190, 191, 197, 23, 319, 36, 369, 400, 470, 606, 643, 690, 78, 789, 80, 89 and 90.



Training files, obtained from the Juliet Test Suite for Java

DATA EXTRACTION Extract methods from the original files- Then, clean, normalize and save them in a new file.

DATA FILTERING Remove non-significant samples and longer sequences

Code	CWE	isVulnerable	one-hot labels
"public void method() {...}"	15	True	[0,0,1,...,0, 1]
"public void method() {...}"	15	False	[0,0,1,...,0, 0]
...	...	...	...

ENCODING & TOKENIZATION Transform code into numerical representations (with *JavaBERT* tokenizer).

TRAIN & EVALUATE Finetune JavaBERT model for multi-label classification. Evaluate model.

OUTPUT Finetuned model for vulnerability detection (multilabel classification).

EXTENSION DEVELOPMENT Integration of the finetuned model in a vscode extension, through a Flask server.

Figure 1: Overview of VDet for Java: summary of the development stages until the proof-of-concept.

and found the latter to yield higher accuracy at the cost of an increased training time [16]. Lastly, we split the dataset: we use 20% of the data for testing and the remaining 80% for training. We use the 20/80 ratio as it is empirically the best division [8].

3.2 Model development

Previous studies either performed binary classification or multiclass classification, assuming classes were mutually exclusive. However, the dataset contains vulnerable and non-vulnerable code, allowing the model to comprehend the difference between constructions and further differentiate between an actual threat and a potential one. In addition, most real-word code files have more than 1 type of CWE in its composition, so it is important to have a model capable of identifying multiple labels at the same time.

Tokenization and encoding: The Hugging Face library[4] provides tokenizers for all their models, and *JavaBERT* is one of them. We use *JavaBERT Tokenizer* to tokenize and encode all code samples from the dataset. Tokenization ensures that the input tokens have the proper length by truncating (if too long) or padding (if too short) the sequence, adding [PAD] tokens when needed. The attention mechanism of a transformer permits the model to ignore these special tokens by analyzing the input ids and corresponding attention mask - an array of 1s and 0s indicating which tokens are padding and which are not. Despite not influencing evaluation measures, like accuracy or f1-score, these tokens are still fully included in all mathematical operations performed by the model, damaging training and evaluation speed. We apply a Uniform Length Batching strategy that sorts the dataset by sequence length and groups samples of similar sizes in batches of 32, reducing the number of tokens by 66% and speeding up training by avoiding redundant computations.

Classify any length sequences: BERT architectures consume 512 tokens max, but it is possible to overcome this limitation with some workaround. Although it is not recommended to divide the sequence into chunks of size 512, add padding and special characters accordingly, and treat each fragment individually during training, it is helpful when classifying longer files. Therefore, we applied this technique to identify vulnerabilities in different sized files.

Model evaluation: For the evaluation of the model, we use the test split of our dataset. For the model's evaluation, we use our dataset's test split. As we are dealing with multi-label classification and the number of samples per label differs, we calculate the weighted average values for precision, recall and f1-score. We further compute each class's False Negative and False Positive rates and calculate their mean values. We highlight the model's accuracy of 98.9%. The results are summarized in Table 2.

Table 2: Performance measures of our model.

Accuracy	Precision*	Recall*	F1-Score*	FNR**	FPR**
0.989	0.95	0.93	0.94	0.071	0.009

* Weighted-averages, ** Mean values

3.3 IDE extension

We selected VS Code to implement the extension, as it is one of the most widely used IDEs [5] and has good support for extension development. In VS Code, the extensions are written mainly in Javascript or Typescript. We develop our tool in the latter and we use the Svelte[6](a lightweight Javascript web framework) for the interface. Since we use Hugging Face's transformer library [22], we developed a Flask[7] server to connect both parties.

Our extension blends neatly into the environment, as illustrated in Figure 2. It has two buttons, "Code Selection" and "Complete File", each with its tooltip for user guidance. On the one hand, when the extension is run for a specific code section defined by the user, the results are presented by line interval. On the other hand, the results are displayed method by method when analyzing the complete file. There is also a progress bar for each label present, and its values correspond to the probability of the labels obtained with the model through classification. These outputs are straightforward, requiring minimal security background and making the extension accessible to most developers.

Scan options: "Code Selection" analyzes a section highlighted by the user in the active text editor. We extract the text from the active selection and pass it to the server for classification, using the line intervals as keys for the selected code. "Complete File" analyzes the complete active file. We extract method names and bodies from the file and hand these data to the server, with the names acting as keys for the method statements. Although the first option allows users to locate vulnerabilities with more precision and faster, it is not the recommended strategy because it is highly dependent on the

[4]https://huggingface.co/

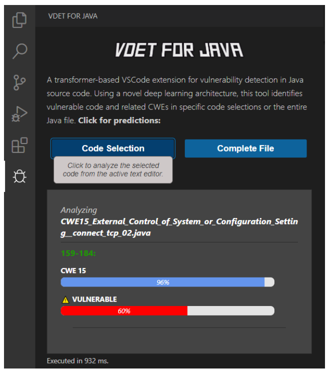
[5]https://pypl.github.io/IDE.html

[6]https://svelte.dev/

[7]https://flask.palletsprojects.com/en/2.1.x/

user's ability to select an appropriate code section. The complete file analysis will likely be more accurate as the extension performs a method-level classification, eliminating the chances of letting a critical code section out of the scan.

Execution Experiment: We performed an experiment to provide a notion of the performance of the execution time of VDet. For that, we select the shortest file and the lengthiest file in our testing dataset, both vulnerable with CWE-400. The former is 40 lines in size and the latter 480 lines, each containing 508 and 3,567 tokens, respectively. We executed VDet five times for each file and obtain an average execution time of 3090ms for the shortest file and 9309ms for the longest.



A transformer-based VSCode extension for vulnerability detection in Java source code. Using a novel deep learning architecture, this tool identifies vulnerable code and related CWEs in specific code selections or the entire Java file. **Click for predictions:**

Click to analyze the selected code from the active text editor.

Analyzing
CWE15_External_Control_of_System_or_Configuration_Setting__connect_tcp_02.java

159-184:

CWE 15 96%

⚠ **VULNERABLE** 60%

Executed in 932 ms.

Figure 2: VDet for Java: example of the outputs of the extension when analyzing a code section.

Flask server as backend: We developed a server with two endpoints to bind the strategies (implemented in Python) to the VS Code extension. For the code selection strategy, the */predict/section* endpoint accepts as input a code selection (with the code and the matching line interval) and outputs the labels and probabilities obtained from the model. The */predict/file* endpoint expects an array of methods (i.e., code and method name) and outputs the labels and probabilities for each one.

4 CONCLUSIONS AND FUTURE WORK

In this paper, we present VDet for Java, which, to the best of our knowledge, is the first transformer-based VS Code extension for vulnerability detection in Java code. The extension can detect up to 21 CWE types, as it incorporates a fine-tuned *JavaBERT* model for multi-label classification. We trained the pre-trained model on a custom dataset with 115,600 methods extracted from the Juliet Test Suite. Our evaluation results indicate an accuracy of nearly 99%, a precision of 95%, and a recall of 93%.

VDet for Java is a successful proof-of-concept extension that provides developers with accurate and accessible vulnerability identifications. We view many potential directions for further improvements. One is improving accuracy by training the model on more data and testing it with non-synthetic samples. We can also complement the identifications with a detailed description of the CWE and potential fix suggestions for the vulnerability. We can expand functionalities to enable the analysis of a project with multiple files and the search for a specific CWE. It is also possible to have faster predictions by optimizing the communication between the extension and the Python-based server.

REFERENCES

[1] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. 2022. How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA.
[2] Leyla Bilge and Tudor Dumitraş. 2012. Before We Knew It: An Empirical Study of Zero-Day Attacks in the Real World. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA.
[3] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning based Vulnerability Detection: Are We There Yet? (9 2020).
[4] Roland Croft, Yongzhen Xie, and Muhammad Ali Babar. 2021. Data Preparation for Software Vulnerability Prediction: A Systematic Literature Review. *ArXiv* abs/2109.05740 (2021).
[5] Nelson Tavares de Sousa and Wilhelm Hasselbring. 2021. JavaBERT: Training a transformer-based model for the Java programming language. *CoRR* abs/2110.10404 (2021). arXiv:2110.10404
[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805
[7] Brian Fitzgerald and Klaas-Jan Stol. 2017. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software* 123 (2017), 176–189.
[8] Afshin Gholamy, Vladik Kreinovich, and Olga Kosheleva. 2018. Why 70/30 or 80/20 Relation Between Training and Testing Sets: A Pedagogical Explanation.
[9] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681.
[10] Xin Li, Lu Wang, Yang Xin, Yixian Yang, and Yuling Chen. 2020. Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning. *Applied Sciences* 10, 5 (2020).
[11] Xin Li, Lu Wang, Yang Xin, Yixian Yang, Qifeng Tang, and Yuling Chen. 2021. Automated Software Vulnerability Detection Based on Hybrid Neural Network. *Applied Sciences* 11 (4 2021), 3201. Issue 7.
[12] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021), 1–1.
[13] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *Proceedings 2018 Network and Distributed System Security Symposium*.
[14] Stuart Millar. 2017. *Vulnerability Detection in Open Source Software: The Cure and the Cause*. Queen's University Belfast.
[15] Justin Smith, Brittany Johnson, Emerson R. Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2019. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Transactions on Software Engineering* 45 (2019), 877–897.
[16] Tim Sonnekalb, Thomas S Heinze, and Patrick Mäder. 2022. Deep security analysis of program code. *Empirical Software Engineering* 27, 1 (2022), 1–39.
[17] Tyler Thomas, Madiha Tabassum, Bill Chu, and Heather Richter Lipford. 2018. Security During Application Development: an Application Security Expert Perspective. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (2018).
[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. (6 2017).
[19] Benjamin White. 2016. Secure Coding Assistant: enforcing secure coding practices using the Eclipse Development Environment.
[20] Michael Whitney, Heather Richter Lipford, Bill Chu, and Jun Zhu. 2015. Embedding Secure Coding Instruction into the IDE: A Field Study in an Advanced CS Course. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015).
[21] Lauren Williams. 2018. Secure Software Lifecycle Knowledge Area. (2018).
[22] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *ArXiv* abs/1910.03771 (2019).
[23] Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton. 2011. ASIDE: IDE support for web application security. In *ACSAC '11*.
[24] Noah Ziems and Shaoen Wu. 2021. Security Vulnerability Detection Using Deep Learning Natural Language Processing. (5 2021).
[25] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* (2019), 1–1.