Detecting Security Patches in Java Projects Using NLP Technology

Andrea Stefanoni[1,2,3], **Šarūnas Girdzijauskas**[2], **Christina Jenkins**[3], **Zekarias T. Kefato**[2], **Licia Sbattella**[1], **Vincenzo Scotti**[1], and **Emil Wåreus**[4]
[1]DEIB, Politecnico di Milano, Via Golgi 42, 20133, Milano (MI), Italy
[2]KTH Royal Institute of Technology, Brinellvägen 8, 114 28 Stockholm, Sweden
[3]Devoteam Creative Tech, Klara Östra Kyrkogata 2B, 111 52 Stockholm, Sweden
[4]Debrided AB , Nordenskiöldsgatan 17, 211 19 Malmö, Sweden
andrea2.stefanoni@mail.polimi.it sarunasg@kth.se
christina.jenkins@devoteam.com zekarias@kth.se
licia.sbattella@polimi.it vincenzo.scotti@polimi.it
emil.wareus@microfocus.com

Abstract

Known vulnerabilities in software are solved through security patches; thus, applying such patches as soon as they are released is crucial to protect from cyber-attacks. The diffusion of open source software allowed to inspect the patches to understand whether they are security related or not. In this paper, we propose some solutions based on state-of-the-art deep learning technologies for Natural Language Processing for security patches detection. In the experiments, we benchmarked our solutions on two data sets for Java security patches detection. Our models showed promising results, outperforming all the others we used for comparison. Interestingly, we achieved better results training the classifiers from scratch than fine tuning existing models.

1 Introduction

The use of *Open Source Software* (OSS) has become a common practice in proprietary projects, especially thanks to the speed up in software production and the costs reduction (Vaughan-Nichols, 2015). However, this practice comes with the risk of introducing vulnerabilities in private code-bases. In this context, we introduce the concept of *security patches*: a security patch is a special type of code patch, which is a set of changes to be applied to some software to update, fix, or improve it. These security patches are designed to solve code vulnerabilities that causes the exposure to cyber-attacks.

The aforementioned code vulnerabilities have been categorised using different notations. The most famous are the *Common Vulnerabilities and Exposures* (CVE) and the *Common Weakness Enumeration* (CWE); both provide a description of the vulnerabilities discovered and the second one is organised hierarchically. Moreover, there exists data bases containing a list of vulnerable commits (changes to the software code base) like the *National Vulnerability Database* (NVD) and the *Software Assurance Reference Dataset* (SARD), which offer an helpful reference to understand these vulnerabilities. They contain examples of vulnerable code paired with the non-vulnerable counterpart, thus providing test cases for software production.

In this work we focus on OSS projects in Java maintained on *GitHub*. On this platform, a commit represent an update to the code base and is composed of two parts: *commit message* (a short description in natural language of the updated piece(s) of code) and *patch* (sometimes called *code changes*, it consists of one or more *hunks*). Hunks are the differences between the old version and the new version of source code files. These hunks are usually surrounded by context lines of the original untouched source code and marked with line numbers. Deleted rows are marked with an initial −, while the added rows start with a +.

Usually, in the process of software development, the software maintainers are overwhelmed by the number of patches released in their dependencies, which can refer to one of those OSS projects. Since applying patches requires extra work and downtime, it is important to prioritise security patches. In this sense, we propose a method based on Natural Language Processing (NLP) technologies to analyse the code modified in the patch, focusing on the semantics expressed in the code, to detect security patches, and thus allow to prioritise them.

We organise this paper as follows: in Section 2 we present the related research works for code analysis and classification, in Section 3 we presents the data sets we used as benchmarks in the experiments, in Section 4 we provide an overview of the models we considered and how we used them to tackle the detection task, in Section 5 we present the experimental approach we followed and the re-

sults we obtained, and in Section 6 we provide final remarks and propose possible future extensions.

2 Related work

NLP is the area of *Artificial Intelligence* (AI) focused on the analysis and synthesis of human language. Recently, the introduction of *Deep Learning*-based techniques in this area has pushed significantly forward the state-of-the-art on many problems. In particular, the development of *Deep Probabilistic Language Models* based on the *Transformer Architecture* (Vaswani et al., 2017) like *GPT* (Brown et al., 2020), *BERT* (Devlin et al., 2019) and *T5* (Raffel et al., 2020) seems to have enabled an impressive step forward. These models for sequence analysis are pre-trained on large text data sets doing simple tasks like next token/word prediction and can be fine-tuned for any problem, yielding impressive results due to the informative hidden representations learnt during pre-training.

These same models and techniques used for natural language, can be also applied for artificial languages, such as programming languages. In fact, according to the *Naturalness Hypothesis* (Hindle et al., 2016; Allamanis et al., 2018), we can treat source code in the same way of a document written in plain natural language. As a result, deep learning models for sequence and graph processing have been actively used to process code, including vulnerability classification (Otter et al., 2018; Semasaba et al., 2020; Wu, 2021).

The application of deep learning techniques to source code analysis evolved similarly to natural language. Early solution tackled the problem of extracting a distributed continuous representation of code pieces similarly to early works for NLP based on embeddings (i.e., vector semantic representations).

Initially, *word embedding* models for NLP used static and shallow embedding matrices to project words into compact and dense representations (Mikolov et al., 2013a,b; Pennington et al., 2014; Bojanowski et al., 2017). Such word representations can be further combined to obtain semantic vectors representing sentences (Pagliardini et al., 2018; Arora et al., 2017; Zhelezniak et al., 2019; Muffo et al., 2021, 2022) or even entire documents (Le and Mikolov, 2014; Chen, 2017; Hosseini et al., 2022).

Following these approaches, *Code2Vec* (Alon et al., 2019) was developed to extract distributed representations of the tokens in a piece of code. However, Code2Vec exploits more complex structures than vanilla word emebedding models, like *Abstract Syntax Trees* (AST), to compute the vector representations.

More recently, models for contextual representation from sequence analysis have emerged: *CodeBERT* (Feng et al., 2020) , for instance, employs the BERT auto-encoder to carry out source code and natural language analysis, serving as impressive feature extraction model that can be used on many downstream tasks, including vulnerability detection. There are also pre-trained models trained directly for patch analysis, like *CommitBERT* (Jung, 2021), however their accessibility is still limited.

Besides feature extraction for code analysis, many works focused also on specific tasks. In the context of security patch detection/classification, many solution work on C/C++ data sets (due to to higher data availability) and employ multiple sub-models to break down the input analysis.

In the case of *SPI* (Zhou et al., 2022) and *PatchRNN* (Wang et al., 2021), both models use multiple *Long Short-Term Memory* (LSTM) (Hochreiter and Schmidhuber, 1997) networks fed using Word2Vec embeddings trained on C code tokens. SPI uses two LSTMs to extract features respectively from the added and deleted lines of code in a patch and it further enhance the input with the commit message to carry out the classification. PatchRNN uses a twin LSTM solution to analyse the code before and after the patch with the information from the commit message to classify the patch. Both models encode the commit message using standard embedding techniques, namely Word2Vec, and use a mixture of experts to combine the results of code analysis with that of the commit messages. Differently, *CC2Vec* (Hoang et al., 2020) processes only the code changes and exploits the hierarchical structure of a patch (divided into *token*, *line*, and *hunk*) thorough a *Hierarchical Attention Network* (HAN). It analyses with two separate networks added and deleted lines and the post-process together the extracted feature vectors. This last approach was employed also for the classification of C language patches to identify the stable ones.

Concerning Java-specific solutions for security patches classification, *Commit2Vec* (Lozoya et al., 2021) represent the closest work to the one we apresent in this paper. However, the data set used by Commit2Vec is only partially available, making

impossible a direct comparison with our work. The Commit2Vec model is based on Code2Vec embeddings: it encodes the AST of previous and current versions code (with respect to the patch), then an attention layer further processes the embedded code differences to perform the classification.

All the aforementioned works use binary classes division to categorise the security patches, while we are also interested in macro-classes identification.

Recent results showed that handcrafted features and a *Random Forest* classifier (Breiman, 2001) are sufficient to obtain reasonable performances on a set of ten macro-classes derived by the original CVE labels (Wang et al., 2020).

3 Data

To the end of this work, we focused only on Java security patches. In particular, we used three separate data sets: the first two are private data collections, while the latter is publicly available and was curated by Ponta et al. (2019).

We merged the first two data sets into one comprising 123 000 samples (i.e., code patches, with 1157 being related to security issues). Samples from the former data set use a binary labelling system, while those from the latter used both CVE and CWE notations. After merging, labels were uniformed to the binary system with the two classes being *security* and *non-security*. The training set was composed of 933 and 918 samples (respectively for the two classes) and the test set was composed of 224 and 239 samples (respectively for the two classes). To cope with the unbalance in the data set we undersampled the non-security class.

The third data set (Ponta et al., 2019) is composed of 1175 security patches labelled with the CVE notation. Due to the high number of different classes, that would have prevented effectively training a classifier, we first converted the CVE notation to CWE (yielding 605 different classes instances) and then we clustered manually the resulting labels down to five:

Improper management of resources patches to solve vulnerabilities connected to resources and variables (e.g., buffer overflow).

Cryptography features patches to solve vulnerabilities connected to data security and information leakage.

Authentication errors patches to solve vulnerabilities connected to access control, authentication, and user sessions.

Other all the security patches that don't fall under the previous categories (e.g., channel errors).

Non-security complementary class to the security patches (e.g., bug fixes, new features, etc.). Samples from this class were taken randomly from the first two data sets.

Pre-processing steps of all data sets consisted in:

- the extraction of added and deleted lines from the patches;
- replacement of comments, strings, and numbers with as many special tokens;
- splitting of function and variable names (we used the most common naming conventions like *snake case*, *camel case*, and *kebab case*);
- deletion of special characters and stopwords (with the exception of java specific ones).

We divided code tokens on spaces and lowercased to all non-special tokens.

4 Methodology

In the following, we describe how we encoded the input sequence representing the code to analyse and the neural network models we considered to carry out the classification task. We distinguished between baseline models, used to get an idea of the performances achievable on the considered data sets, and advanced models, which exploit more complex architectures to obtain the best results.

4.1 Embedding

As happens for natural language, we converted the sequence of tokens written in Java into a continuous vector representation compatible with neural networks. For this task we considered different embedding models:

Uninitialised embeddings we employed 32-dimensional randomly initialised embeddings we trained with the overall models.

Word2Vec we trained a 100-dimensional embedding model on the code contained in the private data sets.

Code2Vec we resorted to a pre-trained model with 128-dimensional embeddings.

Tests showed that uninitialised embeddings yield a better representations for our task. This is also supported by the results we report in Section 5: uninitialised models achieve the best scores.

4.2 Baseline models

We considered two baseline classification models:

XGBoost (Chen and Guestrin, 2016) we trained this model on handcrafted features, similar to those used by Wang et al. (2020), and we employed a count encoder for the patch.

LSTM we employed this baseline similarly to the work on Commit2Vec, we employed this baseline; however, we fed it with the added and deleted lines concatenated with a special separator token.

4.3 Advanced models

As premised, a part from the baselines, we considered more complex models. For many of them we considered a base version and the *patch* version, were the internal model is replicated to analyse separately added and deleted lines as in the work on PatchRNN. We leveraged both pre-trained models coming from previous works or generic uninitialised models:

PatchRNN inspired by the original work, we used twin recurrent networks to encode separately added and deleted lines. We used *Gated Recurrent Units* (GRU) (Cho et al., 2014) with 64 hidden units to build this model.

HAN as for the PatchRNN, we took inspiration from the HAN used in CC2Vec, and implemented a three layer version of it (respectively for word, hunk and file level). In each layer we used GRUs, with 64 hidden units, and attention was computed on top of it. During the hyperparameters search, we fixed the number of files to two and hunks to three for the sake of parallelisation.

CodeBERT we employed a pre-trained transformer trained on source code as it is common practice nowadays in NLP tasks. The input structure is the same of the LSTM baseline. We used both the original pre-trained model and a variant available via the Transfomers library (Wolf et al., 2020) (alternative model link). Additionally, for this model we tested both fine-tuning and simple transfer learning.

PatchCodeBERT we used the pre-trained CodeBERT to build a twin version of it, replicating the initial model and feeding one with the added lines and one with the deleted lines.

Transformer we considered an uninitialised Transformer encoder with bi-directional attention (as BERT), thus re-proposing a smaller version of CodeBERT.

PatchTransformer similarly to what we did with the Transformer and CodeBERT, we used a smaller uninitialised version of PatchCodeBERT that we trained from scratch.

Since many of the models we considered use separate encoders for added and deleted lines in the patches, we developed a merging layer working on the intermediate hidden vectors. The proposed layer, similarly to the one employed by CC2Vec, concatenates the two vectors, their product, their difference, and their cosine and euclidean distances. The resulting vector is passed through a final classification layer. The remaining models directly apply the final projection on the hidden representation.

5 Experiments and results

Table 1: Results on the private data sets.

Method	$\mathbf{F_1}$	
	Validation	**Test**
XGBoost	0.692 ± 0.033	0.695
LSTM	0.823 ± 0.008	0.829
PatchRNN	0.696 ± 0.007	0.635
HAN	0.787 ± 0.007	0.777
CodeBERT	0.767 ± 0.019	0.764
PatchCodeBERT	0.731 ± 0.023	0.728
Transformer	$\mathbf{0.841 \pm 0.014}$	**0.870**
PatchTranformer	0.831 ± 0.014	0.827

Table 2: Results on the data set by Ponta et al. (2019).

Method	macro $\mathbf{F_1}$	
	Validation	**Test**
LSTM	0.661 ± 0.054	0.607
Transformer	$\mathbf{0.667 \pm 0.033}$	**0.635**
PatchTranformer	0.643 ± 0.020	0.601

We divided the experiments following the data sets division. First, we trained multiple models on the private data sets with the binary labelling system. We selected the best models from the first step for training on the third data set with the five macro-categories. To assess the goodness of the results we measured the F_1-*score* achieved by the classifiers on the test and validation sets. The F_1-*score* on the third data set is computed applying macro averaging among the macro-categories. Results on validation sets are reported as *avg.* $\pm$ *std* because we applied 3-fold cross validation.

We reported the results on the private data sets in Table 1. The transformer based solutions clearly outperformed the other models we considered. In this case we employed a 2 layers Transformer network with 32 hidden units, 4 attention heads, and a maximum of 768 tokens in the input sequence. Interestingly Transformer, LSTM, and PatchTransformer models, which achieved the best results, didn't undergo any pre-training, indicating that fine-tuning may be counterproductive in some cases.

We reported the results on the third data set in Table 2. Here we considered only the three best methods from the first experiment. The results confirmed those of the private data sets: the Transformer model performed better than all the other considered solutions. In this case we increased the hidden units size of the Transformer to 128. The drop in the F_1-*score* with respect to the previous experiment was expected since we moved to a multi-class problem where the issue of unbalance has most probably harmed the performances.

6 Conclusion

In this paper we evaluated different approaches for security patches detection in Java OSS using NLP technologies. Despite the general improvements in many NLP tasks due to the use of pre-trained models, in our experiments we found that uninitialised models yield better results than fine-tuned ones; this is most probably due to the insufficient presence of Java code in the pre-training of the considered models. Differently from previous works, we also noticed that using separate sub-models yields worse performances than using a single model.

In the future we are willing to work on two complementary directions. On one side we are interested in exploring other pre-trained models to refine, either sequential or graph ones. On the other side we are interested in working on larger data sets; thus exploiting C/C++ resources can be useful to produce improved models than can be then transferred to under-resourced languages like Java.

References

Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29.

Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A simple but tough-to-beat baseline for sentence embeddings. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching word vectors with subword information. *Trans. Assoc. Comput. Linguistics*, 5:135–146.

Leo Breiman. 2001. Random forests. *Mach. Learn.*, 45(1):5–32.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Minmin Chen. 2017. Efficient vector representation for documents through corruption. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. In *Proceedings of SSST@EMNLP 2014,*

Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014, pages 103–111. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. 2016. On the naturalness of software. *Commun. ACM*, 59(5):122–131.

Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: distributed representations of code changes. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 518–529. ACM.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

Marjan Hosseini, Alireza Javadian Sabet, Suining He, and Derek Aguiar. 2022. Interpretable fake news detection with topic and deep variational models. *CoRR*, abs/2209.01536.

Tae-Hwan Jung. 2021. Commitbert: Commit message generation using pre-trained programming language model. *CoRR*, abs/2105.14242.

Quoc V. Le and Tomás Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org.

Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. 2021. Commit2vec: Learning distributed representations of code changes. *SN Comput. Sci.*, 2(3):150.

Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119.

Matteo Muffo, Roberto Tedesco, Licia Sbattella, and Vincenzo Scotti. 2021. Static fuzzy bag-of-words: a lightweight and fast sentence embedding algorithm. In *4th International Conference on Natural Language and Speech Processing, Trento, Italy, November 12-13, 2021*, pages 176–185. Association for Computational Linguistics.

Matteo Muffo, Roberto Tedesco, Licia Sbattella, and Vincenzo Scotti. 2022. *Static Fuzzy Bag-of-Words: Exploring Static Universe Matrices for Sentence Embeddings*, pages 101–121. Springer International Publishing, Cham.

Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. 2018. A survey of the usages of deep learning in natural language processing. *CoRR*, abs/1807.10854.

Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi. 2018. Unsupervised learning of sentence embeddings using compositional n-gram features. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 528–540. Association for Computational Linguistics.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL.

Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 383–387. IEEE / ACM.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67.

Abubakar Omari Abdallah Semasaba, Wei Zheng, Xiaoxue Wu, and Samuel Akwasi Agyemang. 2020. Literature survey of deep learning-based vulnerability analysis on source code. *IET Softw.*, 14(6):654–664.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Steven Vaughan-Nichols. 2015. It's an open-source world: 78 percent of companies run open-source software.

Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *2021 IEEE Military Communications Conference, MILCOM 2021, San Diego, CA, USA, November 29 - Dec. 2, 2021*, pages 595–600. IEEE.

Xinda Wang, Shu Wang, Kun Sun, Archer L. Batcheller, and Sushil Jajodia. 2020. A machine learning approach to classify security patches into vulnerability types. In *8th IEEE Conference on Communications and Network Security, CNS 2020, Avignon, France, June 29 - July 1, 2020*, pages 1–9. IEEE.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP 2020 - Demos, Online, November 16-20, 2020*, pages 38–45. Association for Computational Linguistics.

Jiajie Wu. 2021. Literature review on vulnerability detection using NLP technology. *CoRR*, abs/2104.11230.

Vitalii Zhelezniak, Aleksandar Savkov, April Shen, Francesco Moramarco, Jack Flann, and Nils Y. Hammerla. 2019. Don't settle for average, go for the max: Fuzzy sets and max-pooled word vectors. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2022. SPI: automated identification of security patches via commits. *ACM Trans. Softw. Eng. Methodol.*, 31(1):13:1–13:27.