DeepTective: Detection of PHP Vulnerabilities Using Hybrid Graph Neural Networks*

Rishi Rabheru
Dept of Computing
Imperial College London
rishi.rabheru16@imperial.ac.uk

Hazim Hanif
Dept of Computing
Imperial College London,
Faculty of Computer Science and Information Technology
University of Malaya, Malaysia.
m.md-hanif19@imperial.ac.uk

Sergio Maffeis
Dept of Computing
Imperial College London
sergio.maffeis@imperial.ac.uk

ABSTRACT

This paper presents DeepTective, a deep learning-based approach to detect vulnerabilities in PHP source code. DeepTective implements a novel hybrid technique that combines Gated Recurrent Units and Graph Convolutional Networks to detect SQLi, XSS and OSCI vulnerabilities leveraging both syntactic and semantic information. Experimental results show that our model outperformed related solutions on both synthetic and realistic datasets, and was able to discover 4 novel vulnerabilities in established WordPress plugins.

ACM Reference Format:
Rishi Rabheru, Hazim Hanif, and Sergio Maffeis. 2021. DeepTective: Detection of PHP Vulnerabilities Using Hybrid Graph Neural Networks. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3412841.3442132

1 INTRODUCTION

The research community has devoted a significant amount of effort to the automated discovery of PHP vulnerabilities, using static, flow, and taint analysis [3, 8] and data mining [12, 18]. These solutions are very efficient in analysing large quantities of code, but tend to suffer from limited detection performance, in terms of false positives or false negatives. Following recent advances in deep learning and natural language processing, security researchers started to develop deep learning approaches to detect software vulnerabilities in C and C++ programs [11, 15]. Only very recently we have seen the first applications of deep learning to PHP vulnerability discovery [5–7]. Both these approaches apply Long-Short Term Memory (LSTM) neural networks to capture non-local dependencies over various transformations of the source code. LSTM is good at finding patterns in sequential data but is not equally well suited to learn from tree- or graph-structured data, which is a more natural representation of program semantics.

In this paper we provide an overview of DeepTective, a machine learning model which detects SQL injection (SQLi), Cross-site scripting (XSS) and Operating system command injection (OSCI) vulnerabilities within PHP source code. In order to learn syntactic and structural properties from source code, DeepTective transforms it into a sequence of tokens to be analysed by a Gated Recurrent Unit (GRU), a neural network related to the LSTM and able to embed sequential information, in this case about the code structure. Novel to our approach for PHP, we attempt to learn semantic properties of source code by analysing the Control Flow Graph (CFG) with a Graph Convolutional Network (GCN), a recent neural network architecture designed to handle graph-like data structures which during training can embed semantic and contextual information of the source code into the classification model. For our best model, this hybrid architecture achieves a 99.92% F1 score on synthetic data from SARD [19], and a 88.12% F1 score on real-world data from GitHub (our own dataset), outperforming related approaches. We tested DeepTective in the wild, and discovered 4 novel SQLi and XSS vulnerabilities in deployed plugins for WordPress.

*An extended version of this paper is available as [14].

2 MODEL OVERVIEW

DeepTective is divided into two key components: a Gated Recurrent Unit (GRU) which operates on the linear sequence of source code tokens, and a Graph Convolutional Network (GCN) which operates on the Control Flow Graph (CFG) of the source code. Each component provides a different mechanism for the model to detect multiple types of vulnerabilities effectively.

2.1 Preprocessing

Our data samples are fragments of PHP code. As a first step, we raise the level of abstraction of the code to a format that will conceptually help the learning process. We extract the linear sequence of parsed tokens in order to capture syntactic dependencies, and we extract the set of intraprocedural CFGs to capture semantic dependencies. We also transform the sequence of tokens and the CFG in a suitable format for consumption by a neural network, that is vectors of numbers. We parse a sample using `phply` [16], a PHP parsing library for Python and obtain an ordered sequence of tokens. In order to focus the learning on a manageable set of interesting tokens, we conflate the long tail of user-defined functions, variables, and constant values into abstract tokens, and retain the concrete token only for selected functions relevant to the vulnerabilities we study, and for the first k instances of variables and values in each sample. For example, we substitute the first 200 variable tokens in a sample with the artificial tokens `VAR0` - `VAR199`, and substitute all

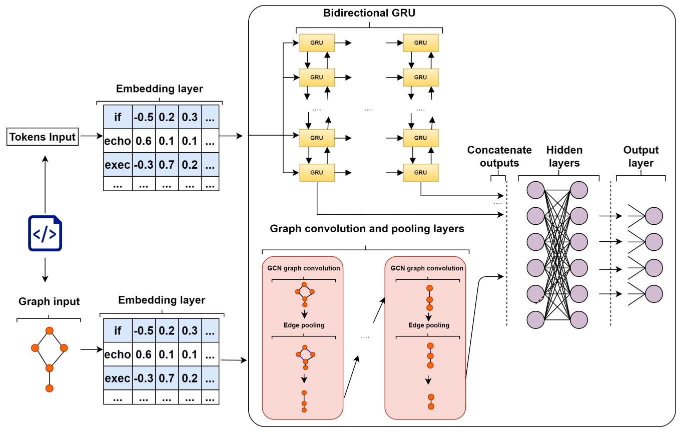

Figure 1: DeepTective architecture

the other ones with the abstract token VAR. We turn each token into a number, using the `LabelEncoder` from `scikit-learn`[17] which, given a vocabulary of tokens, maps each to a sequential natural number.

Next we extract the intraprocedural CFGs from the code, where each graph node represents a line of code, using `joernphp` [1]. To transform the control flow graph into a tensor suitable for consumption by our GCN, we collate the nodes into a 2d matrix where each row contains the features of the node corresponding to the row index.

Figure 1 illustrates the overall architecture of DeepTective. The role of the embedding layer is to transform each numerical input produced in the preprocessing stage into a vector of real numbers, encoding that input as a combination of factors in a lower-dimensional space. We have two embedding layers; one for the token sequence and one for the CFG representation. Both embedding layers use vectors of length 100, which are learned via backpropagation during training and initialised at random. More formally, these layers are simply a mapping from a numerically tokenised function t_i, to a vector $v_i \in \mathbb{R}^{100}$.

2.2 GRU

We extract features from the sequence of tokens representations using a multi-layer bidirectional Gated Recurrent Unit [2] which can learn long term dependencies between the tokens. Source code heavily depends on the syntax of a programming language. Most tokens carry information about the next token in the sequence. This information flow propagates until the end of a code statement. The GRU takes advantage of this information flows by learning bidirectional sequences (i.e. forwards and backwards) of code tokens throughout the source code. The output we take from the GRU is the concatenation of the hidden states at the beginning and end of each layer.

2.3 GCN

The CFG represents the control dependencies of functions and statements in a code sample. Therefore, we extract features from the CFG using a Graph Convolutional Network [9], which is able to embed such dependencies into our model, and learn their significance via backpropagation. Internally we use three layers of a GCN followed by Edge Pooling.

2.4 Classification

We take the output of the graph convolutional layers and flatten it using max pooling. The output of the graph convolutional layers are node vectors of length 4000. The max pooling scans the ith element of each node and selects the maximum values as the ith element of the output vector. We combine the output vector of the GCN with the output vector of the GRU and feed them to the linear classification layers. We have 3 linear classification layers, each with a dropout of 0.3 to combat overfitting, followed by a ReLU activation function. The final output of the ReLU is a probability vector of length 4, representing the confidence of assigning the sample to each class.

3 DATASETS

We built datasets with vulnerable and non vulnerable samples, extracted from synthetic data (SARD) and real-world projects (GIT)

3.1 SARD

Each SARD sample [19] is a short standalone file with no external dependencies. The dataset contains both safe and unsafe samples for XSS, SQLi and OSCI. SARD samples are designed to test security tools, and are rather simplistic and unlikely to reflect code in real world projects. Despite that, SARD has been widely used in previous studies of vulnerabilities, including specifically for PHP [5, 10]. The advantages of SARD are that vulnerabilities are guaranteed to be self-contained in the samples, and each sample has very few irrelevant lines of code, helping to focus the learning process. Besides, the labels of SARD samples are highly accurate, which is a primary concern for supervised machine learning approaches.

3.2 GIT

We also collect a dataset representing vulnerabilities as they actually appear in realistic PHP projects. GitHub hosts source code for PHP projects of all sizes, ranging from the extremely popular WordPress framework to a beginner's first PHP snippet. In order to select representative vulnerabilities, we searched the National Vulnerability Database (NVD) [13] for CVE entries labelled with the CWE identifier 79 (XSS), 89 (SQLi) and 78 (OSCI). We extracted from the references of each relevant CVE any GitHub commit URL, and cloned the corresponding PHP repositories. In combination, we also cloned from GitHub some of the largest and most commonly used open source PHP projects: `Moodle`, `CodeIgniter`, `Drupal`, `ILIAS`, `phppmyadmin`, `wikia`, `magento2`, `simplesamlphp` and `WordPress`. We search the commit history of each cloned project for keywords related to the vulnerabilities we are interested in, including "xss", "sqli" and several variants. There are a few commit messages that report fixing both XSS and SQLi vulnerabilities: we exclude these, as multi-label classification is beyond the scope of this project. When we come across a relevant commit, we extract the vulnerable version of the affected files, and add to each file the label for the corresponding vulnerability. These constitute our positive samples. From the same version of the repository, we save the files not affected by the commit as our negatives samples. Although we expected a fair amount of misslabelling, we manually inspected 10% of the files labelled as positives, and did not detect any mislabelling.

4 EXPERIMENTS

We evaluate DeepTective on data from SARD, GitHub, and from both. This allows us to compare the difference between using synthetic and real-world samples, and investigate the generalisation ability of a model. Furthermore, we compare the classification performance of DeepTective with previous work, and identify interesting variations between the approaches. Finally, we report on new CVEs discovered using DeepTective.

4.1 Experimental Setup

For both experiments, we use Pytorch 1.5 and Torch Geometric 1.5.0 with CUDA 10.1 on top of Python 3.8.1. We train the model on a computer running Intel Xeon Skylake CPU (40 cores), 128GB RAM and Nvidia GTX Titan XP. Throughout the experiments, we report true negatives (TN), false negatives (FN), true positives (TP), false positives (FP), accuracy, precision, recall and F1-score. Accuracy measures the percentage of correctly predicted samples, but is not very significant when test classes are imbalanced, like in the case of vulnerabilities which are very rare compared to safe samples. A trivial classifier marking everything as safe would have very high accuracy in the real world, but little use. Precision measures how many of the reported vulnerabilities are actual vulnerabilities: it tells us if it is worth investigating the results of the classifier. Recall measures the percentage of existing vulnerabilities that the classifier is able to discover: it tells us how worried we should still be after running the tool. Finally, the F1 score summarises numerically the balance between precision and recall.

Since this is a multiclass-classification problem, we use cross-entropy as our loss function. The training process uses a batch size of 64 along with an Adam optimiser and a learning rate of 0.00001. Alongside this, we implement a learning rate scheduler that reduces the learning rate if the loss plateaus. Lastly, we split the dataset for training/validation/test to 80/10/10, and stratify data according to their classes. With the model and hyper-parameters in place, we train the model for 150 epochs to maximise the learning potential of our model.

4.2 Classification Performance

Table 1 shows the result of training our model and testing on SARD, GIT and their combination ALL. We report only the figures for the binary classification problem where the positives classes XSS, SQLi and OSCI are merged in the Unsafe class. This is to simplify exposition, and because ultimately we care mostly about detecting vulnerabilities, irrespective of their specific label.

The File-S model, which is trained on SARD, achieves perfect scores for all the metrics when testing on the SARD dataset itself. The results for for GIT are disappointing, showing that patterns learned from the synthetic dataset do not transform to realistic projects. The performance on the combined datasets is roughly a weighted average of the other two.

A substantial improvement instead is observed on the performance of the File-G model, in particular on GIT (the dataset it is

Table 1: Classification: Vulnerable (TP) or Safe (TN).

Model	Testing set	TN	FN	TP	FP	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
File-S (SARD)	SARD	1624	0	589	0	100	100.0	100.0	100.0
	GIT	1817	2263	465	909	36.89	33.84	17.05	22.67
	ALL	3439	2263	1054	911	55.13	53.64	31.78	39.91
File-G (GIT)	SARD	9143	2010	3878	7097	54.62	35.33	65.86	45.99
	GIT	251	44	229	22	83.33	91.24	83.88	87.40
	ALL	9396	2054	4107	7117	55.32	36.59	66.66	47.25
File-A (ALL)	SARD	1624	1	588	0	99.95	100.0	99.83	99.92
	GIT	240	32	241	33	82.78	87.96	88.28	88.12
	ALL	1864	34	828	33	96.56	96.17	96.06	96.11

Table 2: Comparison: DeepTective File-A vs. Selected Tools.

Tool name	TN	FN	TP	FP	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
A: Results for SARD dataset.								
DeepTective	1624	1	588	0	**99.95**	**100.0**	99.83	**99.92**
TAP	1584	96	493	40	93.85	92.50	83.70	87.88
wirecaml-XSS	470	50	385	1308	38.64	22.74	88.51	36.18
wirecaml-SQLi	1496	0	91	626	71.71	12.69	**100.00**	22.52
progpilot	629	304	285	995	41.30	22.27	48.39	30.50
WAP	1342	477	112	282	65.70	28.43	19.02	22.79
RIPS	1440	497	92	184	69.23	33.33	15.62	21.27
B: Results for GIT dataset.								
DeepTective	240	32	241	33	82.78	**87.96**	**88.28**	**88.12**
TAP	233	262	11	40	44.69	21.57	4.03	6.79
wirecaml-XSS	299	171	41	35	62.27	53.95	19.34	28.47
wirecaml-SQLi	484	60	0	2	**88.64**	0.00	0.00	0.00
progpilot	265	257	16	8	51.47	66.67	5.86	10.77
WAP	160	154	119	113	51.10	51.29	43.59	47.13
RIPS	256	225	48	17	55.68	73.85	17.58	28.40

trained on). It achieves 91.24% precision and 83.88% recall, for an F1 score of 87.40%. This shows that although GIT is a harder dataset to learn, consisting of highly diverse PHP files from popular projects, training on realistic samples greatly improves performance.

Training on the combined dataset has the effect of slightly reducing the perfect performance of File-S on SARD, but yields a larger increases over the F1 score of File-G on GIT. In particular, File-A finds more real-world vulnerabilities (increase of TP) but at the price of a few more false alarms (increase of FP).

4.3 Tool Comparison

We compared the performance of DeepTective File-A, our best model, with selected publicly available tools to find PHP vulnerabilities, based on machine learning (`wirecaml` and `TAP`) or static analysis (`progpilot`, `RIPS` and `WAP`) [3–5, 10, 12]. We measured the tools detection performance, which is reported in Table 2. Note that `wirecaml` is made of two binary classifiers for XSS and SQLi and thus we report the performance of each individual classifier. Furthermore, vulnerabilities of the class that is not being classified by a `wirecaml` classifier were deemed as safe samples when judging performance.

The results show that DeepTective significantly outperformed the other tools in terms of F1 score. `TAP` achieved a high F1 on the synthetic SARD dataset, but poor performance on the realistic samples from GIT. `wirecaml-SQLi` achieved a high accuracy on GIT, but at the price of null precision and recall. Note that the same tool had perfect recall on the SARD dataset. On a synthetic dataset intersecting with with our SARD, [5] measured F1 scores of 98.8%

and 97.5% for TAP and `wirecaml` respectively. Our failure to replicate a similar result for those (pre-trained) tools on SARD points to the difficulty for some machine learning models to generalise even to related datasets. We have noted above how a perfect 100% F1 for File-S on SARD translated into a poor 22.67% F1 for the same model on GIT. That result is in line with the drop observed in the performance of all the tools above from testing on SARD to testing on GIT. We believe our results show that evaluating tools only on synthetic datasets is not a sufficient guarantee of practical performance, and that DeepTective File-A stands out in its ability to perform well on realistic samples.

4.4 Case Studies

We tested if DeepTective could be used to detect novel vulnerabilities in deployed PHP projects. We selected 5 WordPress plugins with a limited number of users (less than 20,000), to increase the likelihood of them containing an undiscovered security vulnerability. Projects with a limited user base may have a smaller development team lacking security expertise, or be subject to less scrutiny than popular projects.

In absence of ground truth, we need to resort to manual inspection to verify the results. We manually inspected 277 reported positives. Several appeared suspicious but we did not have sufficient familiarity with the application to determine unequivocally if they constituted a vulnerability. We were able to confirm 4 of these as actual security vulnerabilities, and we responsibly disclosed our findings to the respective developers. We publicly disclosed 2 of them after they were patched: **CVE-2020-14092** and **CVE-2020-13892**. We tested the tools from Section 4.3 on these projects to see if they could detect either of the above vulnerabilities, but none succeeded.

5 CONCLUSIONS

We have presented DeepTective, a novel vulnerability detection approach which aims to capture contextual information from real-world vulnerabilities in order to reduce false positives and false negatives. Our approach combines a Gated Recurrent Unit to learn long term sequential dependencies of source code tokens and a Graph Convolutional Network to incorporate contextual information from the control flow graph. DeepTective achieves a better performance that the state-of-the-art on both synthetic and realistic datasets, and was able to detect 4 novel security vulnerabilities in WordPress plugins, which other detection tools failed to detect. For further details, see the full version of this paper [14].

REFERENCES

[1] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*.

[2] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H.r Schwenk, and Y. Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv:cs.CL/1406.1078

[3] J. Dahse and J. Schwenk. 2010. RIPS-A static source code analyser for vulnerabilities in PHP scripts. In *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*.

[4] designsecurity. [n. d.]. progpilot. https://github.com/designsecurity/progpilot [Online; accessed June 05, 2020].

[5] Y. Fang, S. Han, C. Huang, and R. Wu. 2019. TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology. *PLOS ONE* 14, 11 (2019).

[6] A. Fidalgo, I. Medeiros, P. Antunes, and N. Neves. 2020. Towards a Deep Learning Model for Vulnerability Detection on Web Application Variants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

[7] N. Guo, X. Li, H. Yin, and Y. Gao. 2020. VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode. In *Information and Communications Security*. Springer International Publishing.

[8] N. Jovanovic, C. Kruegel, and E. Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S P'06)*.

[9] T. Kipf and M. Welling. 2016. Semi-supervised classification with graph convolutional networks. (2016). arXiv:cs.CL/1609.02907

[10] J. Kronjee, A. Hommersom, and H. Vranken. 2018. Discovering Software Vulnerabilities Using Data-Flow Analysis and Machine Learning. In *Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES 2018)*.

[11] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[12] I. Medeiros, N. Neves, and M. Correia. 2014. Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14)*.

[13] National Institute of Standards and Technology. [n. d.]. National Vulnerability Database. https://nvd.nist.gov/ [Online; accessed April 10, 2020].

[14] R. Rabheru, H. Hanif, and S. Maffeis. 2020. A Hybrid Graph Neural Network Approach for Detecting PHP Vulnerabilities. arXiv:cs.CR/2012.08835

[15] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*.

[16] S. Pitucha. [n. d.]. phply. https://github.com/viraptor/phply [Online; accessed April 18, 2020].

[17] scikit-learn. [n. d.]. LabelEncoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html [Online; accessed April 18, 2020].

[18] S. Son and V. Shmatikov. 2011. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS '11)*.

[19] B. Stivalet. [n. d.]. PHP Vulnerability Test Suite. https://samate.nist.gov/SARD/view.php?tsID=103 [Online; accessed April 10, 2020].