A Hybrid Graph Neural Network Approach for Detecting PHP Vulnerabilities[*]

Rishi Rabheru[§]
Department of Computing
Imperial College London, UK
rishi.rabheru16@imperial.ac.uk

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

Sergio Maffeis
Department of Computing
Imperial College London, UK
sergio.maffeis@imperial.ac.uk

***Abstract*—We validate our approach in the wild by discovering 4 novel vulnerabilities in established WordPress plugins. This paper presents DeepTective, a deep learning-based approach to detect vulnerabilities in PHP source code. Our approach 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. We evaluate DeepTective and compare it to the state of the art on an established synthetic dataset and on a novel real-world dataset collected from GitHub. Experimental results show that DeepTective outperformed other solutions, including recent machine learning-based vulnerability detection approaches, on both datasets. The gap is noticeable on the synthetic dataset, where our approach achieves very high classification performance, but grows even wider on the realistic dataset, where most existing tools fail to transfer their detection ability, whereas DeepTective achieves an F1 score of 88.12%.**

***Index Terms*—Vulnerability detection; PHP vulnerabilities; Graph neural networks; Software security.**

I. Introduction

PHP remains the most common server-side language on the web, especially among the long tail of medium and small size websites. Due to the amount of economic activity taking place online, PHP web applications remain a tempting target for malicious actors looking to exploit security vulnerabilities for financial gain or in pursue of other illicit ends.

In order to prevent the compromise of PHP web applications there has been a steady and growing trend by developers, security firms and white hat hackers to find, fix and disclose PHP vulnerabilities [2]. The research community has also devoted a significant amount of effort to the automated discovery of PHP vulnerabilities, leveraging static, flow, and taint analysis techniques [3]–[5] as well as data mining approaches [6]–[10]. 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]–[13]. Only very recently we have seen the first applications of deep learning to PHP vulnerability discovery [14]–[16]. Both these approaches apply Long-Short Term Memory (LSTM) neural networks to capture non-local dependencies over various transformations of the source code.

In this paper we present DeepTective, a deep-learning based vulnerability detection approach, which aims to combine both syntactic and semantic properties of 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 syntactic structure. Novel to our approach for PHP, we attempt to learn semantic properties of the source code by analysing the 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 (SARD) and a 88.12% F1 score for real-world data (our own dataset).

We investigate the impact of different dataset distributions for detecting multiple vulnerabilities, and the challenges in creating such datasets. The key dimensions to take into account are the nature of the samples (synthetic versus realistic), the accuracy of the labels, the balance of the classes and the overarching difficulty in generating high-quality datasets. We systematically compare the performance of DeepTective and a number of existing PHP vulnerability detection tools on SARD and on our real-world dataset. DeepTective outperforms the other tools on both datasets, but the gap is significantly wider on the real-world one, even for pre-trained models. Finally, we tested DeepTective in the wild, evaluating its execution performance and its ability to generalise to a number of real-world PHP applications not present in the training dataset. We validated the practical usefulness of DeepTective by discovering 4 novel SQL injection and Cross-site scripting vulnerabilities in deployed plugins for WordPress.

In summary, our main contributions are:

[*]A preliminary version of this paper was presented as a poster in [1].
[§]This author was affiliated with Imperial College London during most of the work for this paper.

- The first investigation of the use of GCN and GRU to detect vulnerabilities in PHP source code, embedding both syntactic, structural and semantic information in the machine learning model;
- An analysis of the impact of dataset definition on model performance for vulnerability discovery, and the collection of file-level labelled PHP datasets, which we make available to the public;
- An extended evaluation of DeepTective, our GNN, by comparing it with selected existing tools for PHP vulnerability detection and by using it in the wild, where we discovered 4 novel vulnerabilities in established WordPress plugins.

II. Background

In this section, we survey automated vulnerability detection approaches for PHP source code and discuss recent advances in using graph neural networks for vulnerability detection.

A. Detecting Vulnerabilities in PHP

Researchers and practitioners, over the years, have developed many tools to detect vulnerabilities in PHP applications.

1) Traditional Approaches: Traditional approaches focus on the use of static, semantic and taint analysis to locate vulnerabilities. Pixy [3] implements flow-sensitive and context-sensitive data flow analysis to detect vulnerable components in a PHP web applications, mainly targeting XSS. RIPS [4], [5] combines taint and static analysis to locate vulnerable program points in a PHP application. However, RIPS and Pixy are unable to analyze flaws that require the analysis of multiple files. phpSAFE [9] performs a lexical and semantic analysis of code at the Abstract Syntax Tree (AST) level, before executing an inter-procedural analysis to follow the flow of tainted variables starting from the `main` function. Differently from previous approaches, SAFERPHP [6] focuses on the detection of Denial of Service (DoS) and missing authorisation checks. It also performs inter-procedural and semantic analysis by analysing the control dependencies via the control flow graph (CFG).

2) Data Mining Approaches: More recent approaches aim to detect PHP web application vulnerabilities using data mining techniques. WAP [7], [8] implements taint analysis along with a number of machine learning models to predict vulnerable PHP samples. Logistic Regression obtains the best performance, and is able to detect 8 classes of vulnerabilities, including SQLi, XSS, and OSCI. In a follow up work, DEKANT [17] adopts Natural Language Processing (NLP) techniques to detect vulnerabilities. In particular, it uses a Hidden Markov Model (HMM) [18] to characterise vulnerabilities based on a set of source code slices. These code slices are marked as tainted or non-tainted and then passed on for further analysis. WIRECAML [19] combines data-flow analysis and machine learning to detect SQLi and XSS vulnerabilities in PHP source code. The combination of reaching definition, taint and reaching constant analysis allows the tool to extract meaningful data flow features from the CFG, and optimise the learning process of the machine learning model.

3) Deep Learning Approaches: More recently, deep learning is being applied to vulnerability detection for PHP source code. TAP [14] extracts code tokens from PHP codes using a custom tokenizer, and performs data flow analysis to find relevant lines of code that contain function calls. TAP uses Word2Vec to generate numerical vectors from the code tokens, and implements a sequence-based deep learning technique called Long Short-term Memory (LSTM) to train the detection model. Vulhunter [16] proposes a different approach leveraging bytecode features to represent vulnerabilities. Vulhunter generates CFGs, data-flow graphs (DFGs) and analyses them to generate potentially suspicious code slices. The code slices are transformed into bytecode slices. Like TAP, Vulhunter uses Word2Vec to generate vectors from the bytecode slices and passed to a Bi-directional LSTM. Also [15] leverages PHP bytecode to locate vulnerabilities. Code slices are translated to bytecode using the Vulcan Logic Dumper (VLD), which intercepts Zend bytecode before it executed. The authors train a 2-layer LSTM model and achieved 95.35% accuracy, 96.51% precision and 96.14% recall.

B. Graph Neural Network for Vulnerability Detection

Recently, Graph Neural Networks (GNNs) have been applied to vulnerabilities detection in source code.

Devign [11] implements a Gated Graph Recurrent Network with Conv layer to embed the information from the CFGs and DFGs of C/C++ programs. The composite graph representation enables the model to learn contextual information from source code effectively. This contextual information helps the model understand parts of the program behavior, such as sematic dependencies, which are relevant to detecting vulnerabilities. Similarly, FUNDED [20], leverages Gated Graph Neural Network to capture and reason about a program's control, data, and call dependencies. It learns the program representation from the graph structures of real-world C and Java source code. Evaluation results showed that FUNDED is better at detecting vulnerabilities than previous approaches such as Devign [11], VulDeePecker [13] and µVulDeePecker [21].

PHP is an interpreted language, and its semantics is substantially different from C/C++ or Java [22]. To the best of our knowledge, we are the first to investigate the role that GNNs can play in the analysis of PHP code.

III. DeepTective

In this Section, we introduce DeepTective, our novel PHP vulnerability detection model. DeepTective detects SQLi, XSS and OSCI vulnerabilities within source code at a file-level granularity. It 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. We combine the GRU and GCN in a novel hybrid architecture able to leverage the strengths of both techniques.

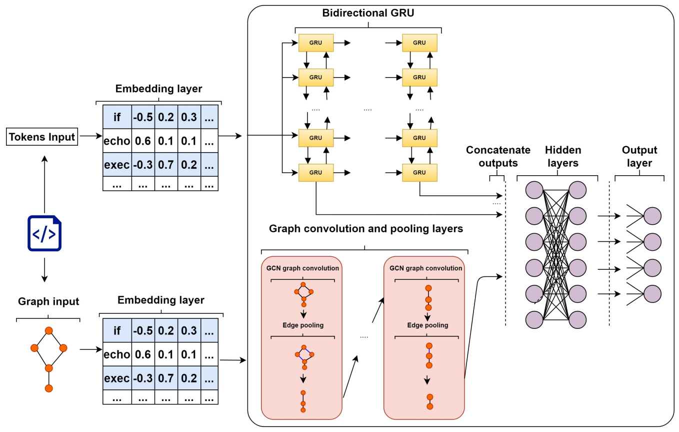

Fig. 1: DeepTective architecture

A. Preprocessing

Our data samples are PHP files. 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.

1) Sequence of Tokens: We parse a sample using `phply` [23], a PHP parsing library built on top of `ply` [24], an implementation of the `yacc` and `lex` parsing tools for Python. From parsing, we obtain an ordered sequence of tokens. We remove tokens for comments, tabs, spaces, and PHP open and close tags from the sequence, as the presence or absence of a vulnerability is not affected by these.

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 the first k instances found in each sample. We substitute the first 200 variable tokens in a sample with the artificial tokens VAR0 - VAR200, and substitute all the other ones with the abstract token VAR. We also retain the concrete token for selected PHP functions such as `query`, `exec`, `strip_tags` which are relevant to the vulnerabilities we study, and typically represent sinks or santizers.

Next, we turn each token into a number, using the `LabelEncoder` from `scikit-learn` [25] which, given a vocabulary of tokens, maps each to a sequential natural number. The GRU that consumes our token sequences requires vectors of fixed length as inputs. Based on empirical observations on the training set, we chose a fixed maximum length of 3000 tokens per sample.

2) CFG: We use `joernphp` [26] to parse and extract the CFGs from each sample, as it proved robust even for large files. We use the same procedure as for sequences of tokens above, but with a fixed length of 20, to turn each CFG node into a numerical vector. Next we represent the CFG edges as a vector of tuples (i, j) representing a directed edge from node i to node j.

B. Model Architecture

Figure 1 illustrates the overall architecture of DeepTective. We now describe each component and summarize the architectural parameters.

1) Embedding Layer: 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 higher-dimensional space. We have two embedding layers of size 100; one for the token sequence and one for the CFG representation. 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) GRU: We extract features from the sequence of tokens representations using a multi-layer bidirectional Gated Recurrent Unit [27] which can learn long term dependencies between the tokens. Code patterns, such as those leading to vulnerabilities, heavily depend on the syntax of a programming language and the local context in which they appear. We use 3 layers of GRU and internally, each layer of the GRU computes the following function for each element in the input sequence:

$$\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}$$

where h_t is the hidden state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0, and r_t, z_t, n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product. The output we take from the GRU is the concatenation of the hidden states at the beginning and end of each layer.

3) GCN: The CFG represents the control dependencies of functions and statements in a code sample. These approximate the flow of information from untrusted sources to sensitve sinks typical of injection vulnerabilities. Therefore, we extract features from the CFG using a Graph Convolutional Network [28], 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. Let $\mathbf{X}$ be a graph node vector, and $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ the adjacency matrix of the graph, with inserted self-loops. The equation $\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\ \hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta}$ defines the convolved signal matrix $\mathbf{X}'$, where $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ denotes the diagonal degree matrix and $\mathbf{\Theta}$ denotes the convolutional filter parameters [29].

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.

The intuition behind why we selected GRU and GCN as our main model for DeepTective is that we would like an architecture that can learn both long-term dependencies from source code tokens and source code contextual semantics at the same time. We can achieve this by combining these two architectures into a single neural network model that simultaneously learns the respective features.

IV. Datasets

In order to evaluate a supervised vulnerability detection model, we need to build datasets with vulnerable and non vulnerable samples.

We label the samples as Safe, XSS, SQLi and OSCI, where the latter 3 labels together are the Unsafe "virtual" label. We extract the samples from synthetic data (SARD) and real-world projects (GitHub), as detailed below. In order to support further research in the area, and facilitate the comparison between different approaches, we make our datasets available to the public[1].

A. Synthetic Samples

The Software Assurance Reference Dataset project [30] is a collection code samples for multiple programming languages. Below, we consider the subset of SARD for PHP vulnerabilities [31]. Each sample is a short standalone file with no external dependencies. Samples are generated by a tool called the PHP Vulnerability Test Suite Generator [32]. The dataset contains both safe and unsafe samples for different vulnerability types. 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. This helps focusing the learning process. We extract the PHP code from each SARD safe and unsafe sample for XSS, SQLi and OSCI. We denote by SARD$^{\#}$ our derived dataset. The number of samples in the original SARD dataset and in our dataset are reported in Table I.

B. Realistic Samples

Besides the focused, synthetic samples from SARD$^{\#}$, we want to 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) [33] for CVE entries labelled with the CWE identifier of XSS (CWE-79), SQLi (CWE-89) and OSCI (CWE-78). 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`.

[1]The dataset URL will be released upon publication of this paper.

TABLE I: Number of Samples in Relevant Datasets.

Dataset	Safe	XSS	SQLi	OSCI
SARD	16240	4352	912	624
SARD$^{\#}$	2928	960	288	250
GIT	2726	2117	604	7

1) Sample Extraction: 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.

2) Label Noise: The approach described above may introduce noise in the labelling of samples. Files may be mislabelled when a commit message misidentifies a vulnerability. Vulnerable files with a commit message that does not mention a vulnerability fix, and files which contain vulnerabilities not known or fixed by the developers, will be mistakenly labelled as negatives. A vulnerability-relevant commit may also include unrelated changes to non-vulnerable files. These files will be mistakenly labelled as positives. To limit these effects, we ignore commits modifying more than 20 files, and we discard changes that only consist in deleting lines of code, as both cases are mostly associated with code refactoring. We manually inspected 10% of the files labelled as positives, and did not detect any mislabelling. We denote our dataset by GIT. The number of samples of each class in GIT are reported in Table I.

V. Model Evaluation

We evaluate DeepTective on its ability to classify files as containing at least one vulnerability, or none. For this task, we train and test the model on data from SARD, GitHub, and from both. This allows us to compare the difference between using synthetic and real-world samples. Furthermore, we compare the classification performance of DeepTective with previous work, and identify interesting variations between the approaches.

A. Methodology

1) Experimental Setup: For this experiment, 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.

2) Performance Criteria: For each experiment, we report true negatives (TN), false negatives (FN), true positives (TP), false positives (FP), accuracy, precision, recall and F1-score.

Note that in Table II we report only the figures for the binary classification problem where the positives classes XSS,

TABLE II: Evaluation results on different models across different datasets.

Model	**Testing set**	**TN**	**FN**	**TP**	**FP**	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1 (%)**
File-S	SARD$^{\#}$	1624	0	589	0	100	100.0	100.0	100.0
(SARD$^{\#}$)	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	SARD$^{\#}$	9143	2010	3878	7097	54.62	35.33	65.86	45.99
(GIT)	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	SARD$^{\#}$	1624	1	588	0	99.95	100.0	99.83	99.92
(ALL)	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

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.

3) Model Training: 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 10^{-5}. 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.

B. Classification

We perform several experiments to investigate different learning patterns across synthetic and real-world source codes. Table II shows the result of training and testing our model on SARD$^{\#}$, GIT and their combination ALL.

File-S. The File-S model is trained on the SARD$^{\#}$ dataset. The precision, recall and F1 scores are 100%. This means that all vulnerable and non-vulnerable PHP samples are correctly classified as positives and negatives. However, File-S fails spectacularly on the real-world GIT dataset, with precision and recall down respectively to 33.84% and 17.05%. We hypothesize that this failure to generalise is due to the highly skewed and homogeneous nature of SARD$^{\#}$ samples on which the model is trained. In particular, the model fails to detect most of the vulnerable GIT samples (2263 FN). On inspection, SARD$^{\#}$ vulnerable samples are short and focused around the vulnerability, whereas GIT vulnerable files may contain a lot of irrelevant context, and more varied vulnerability patterns. As can be expected, the performance on ALL is roughly a weighted average of the preceding two.

File-G. The results for the File-G model are qualitatively similar, but the performance on the SARD$^{\#}$ dataset is disappointing in absolute terms, with 35.33% precision, 65.86% recall and 45.99% F1. We believe this shows that file-level model is appropriate for real world code. Next, we investigate if combining SARD$^{\#}$ and GIT could introduce synergies which improve the classification performance.

File-A. Training on the combined dataset has the effect of slightly reducing the perfect performance of File-S on SARD$^{\#}$by 1 FN, but yields larger increase over the recall and F1 scores 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). Finally, note that the jump in performance on ALL is mostly an artefact of the lower number of samples available for testing, as 90% of both SARD$^{\#}$ and GIT data is used for training. This leads to a higher weight given to the SARD$^{\#}$ performance in comparison to the File-S case. Figure 2 compares the percentage of correct predictions for each fine-grained class on the ALL test set, for File-S,-G and -A.

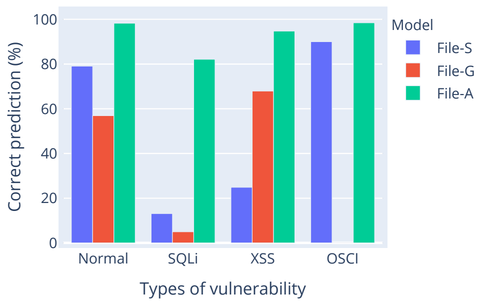

Fig. 2: Distribution of correctly predicted samples across different types of vulnerability.

The average predictive capability of File-A is higher than 80% for all classes. We believe this shows that File-A model is appropriate for real world codes. In fact, by manually inspecting GIT samples we can observe that although a vulnerability may in effect be present inside a function, the vulnerable line by itself is not sufficient to detect the function as vulnerable. As an extreme example, the identify function can be considered as a vulnerable instance of a function to sanitize user input: but inspecting the identity function by itself gives no clues to the presence of a vulnerability. Hence the additional contextual information provided at the file-level has a significant impact also at the multi-class classification level.

C. Tool Comparison

We compared the classification 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`) [4], [7], [14], [19], [34]. We ran all the tools above on the same test sets from the SARD$^{\#}$ and GIT datasets which we used in Section V-B to evaluate File-A. We measured the tools detection performance, which is reported in Table III. 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. Machine learning tools often perform better when trained and tuned using their authors' datasets. Hence, we used `wirecaml` and `TAP` trained

TABLE III: 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

on their respective datasets, effectively testing their ability to generalise to new datasets.

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 showed poor performance on the realistic samples from GIT. wireacaml-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$^{\#}$, [14] measured F1 scores of 98.8% and 97.5% for TAP and wireacaml 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, except for RIPS and WAP. Surprisingly, RIPS and WAP gained more performance based on recall and F1 scores on GIT. However, their performance is still significantly low to be use in the wild. 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.

VI. Practical Experiments

In order to evaluate the practical usefulness of our model, we ran it on a number of PHP projects which we did not include in our GIT dataset. In particular we want to estimate the execution performance, to ensure that the tool can scale also to large projects, and assess it usability for actual vulnerability detection.

For these experiments we have chosen 13 software projects divided in two sets: 8 **popular projects** and 5 **smaller plugins**. The popular projects are listed in the top 50 GitHub repositories (based on stars), that use PHP as their primary language. These are meant to be a representative benchmark for the execution performance. We expect the popular projects to be carefully reviewed, hence we make the assumption that they currently have no security vulnerabilities, and we assume no TP and FN for classification purposes. We also collect 5 WordPress plugins projects, 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. Below we report the execution performance and accuracy for both sets, then we dig deeper on the smaller plugins sets to hunt for vulnerabilities, to limit the effort necessary in manually reviewing positives.

A. Execution Performance

The size of the software projects considered varies from 110KB with 2713 lines of codes (LoC) to 27MB with 242,299 lines of codes. The size and LoC distribution of these software projects reflect the distribution of real-world projects as some projects are small and large in scale.

To evaluate the execution performance of DeepTective across real-word software projects, we use the following performance metrics: (i) **Lines of codes (LoC)** - The number of lines of codes in each file for all the PHP files in a specific software project; (ii) **Processing time** - The time taken (in seconds) to process and transform a PHP file to the data structure used by our detection model; (iii) **Inference time** - The time take to perform the classification of all the PHP files in a specific software project; (iv) **Time/LoC** - The average total time taken (processing and inference) per line of code for a software project; (v) **Time/File** - The average total time taken (processing and inference) per file in a software project.

Table IV-A shows the execution performance for popular projects. Symphony has the longest processing time of 1699.91 seconds as it has the most number of PHP files and LoC. Laravel has the shortest processing time of 17.12 seconds, despite having a higher number of LoC (2713) than PHPMailer (2185). This is due to the simpler strucutre of Laravel code, which is a lightweight PHP framework containing the wireframe to develop a PHP web application. In terms of inference time, the data shows a consistent trend based on the number of PHP files in a project. The higher the number of PHP files, the longer the time it takes to perform inference, as the process is done on the file-level granularity. Time/Loc metric demonstrates minor differences across all the software projects in Github. However, the Time/File metric shows some surprising pattern as Composer has the highest execution time per file even though the number of total PHP files and execution time are lower than other larger projects like Symphony and CodeIgniter. Composer [35] is a dependency management tool for PHP projects, which allows the user to declare, update and manage external libraries. Based on this, it shows that the complexity of Composer contributes to the high

TABLE IV: DeepTective Execution Performance on Real-World Software Projects

Software project	Size (bytes)	PHP files	LoC	Processing time (s)	Inference time (s)	Time (s)/LoC	Time (s)/File	File-A accuracy(%)	File-G accuracy(%)
A: Results for popular projects									
Codeigniter	7,416,704	669	138495	728.4836	7.4599	0.00531	1.10006	53.81	56.35
Composer	2,342,547	252	53518	384.9617	3.1456	0.00725	1.54011	55.16	72.22
Grav	5,955,146	347	60922	400.0879	4.0205	0.00663	1.16458	55.62	62.25
Guzzle	352,741	32	4555	28.1737	0.7210	0.00634	0.90296	50.00	59.38
Laravel	110,595	53	2713	17.1215	0.8413	0.00662	0.33892	69.81	75.47
PHPMailer	381,439	55	2185	20.6959	0.9937	0.00993	0.39436	96.36	96.36
PHPUnit	1,373,437	323	35367	225.5044	3.8504	0.00648	0.71008	66.80	83.59
Symphony	27,052,061	2676	242299	1699.9081	26.2258	0.00712	0.64504	75.85	75.67
B: Results for smaller plugins									
Appointment Booking Calendar	2,826,657	16	4735	50.1272	0.8017	0.01076	3.18306	31.25	56.25
Payment Form for PayPal Pro	1,005,490	13	4379	44.5420	0.7712	0.01035	3.48563	15.38	53.85
PayPal for Digital Goods	149,137	7	1152	6.1942	0.5617	0.00586	0.96514	57.14	42.86
Sportpress	4,834,097	256	50461	419.3428	3.4818	0.00838	1.65166	50.39	48.05
Simple Jobs Board	9,783,895	198	19775	108.8408	2.3262	0.00562	0.56145	86.87	86.36
Total	63,583,946	4897	620556	4133.9838	55.2009	0.00675	0.85546	61.98	71.37

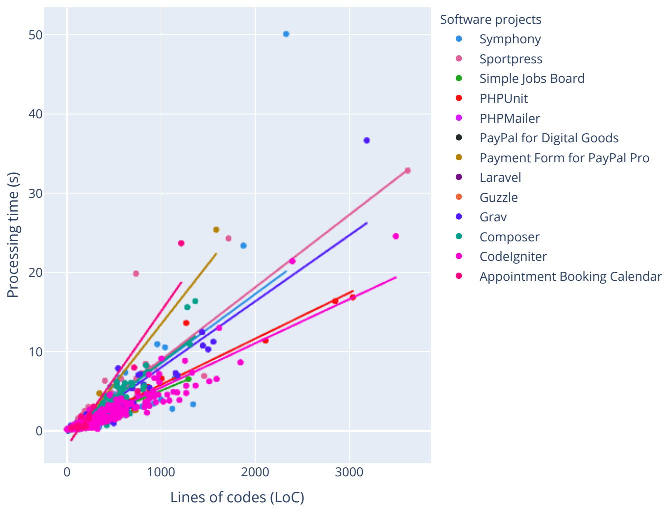

Fig. 3: Processing Time over LoC for all Software Projects

Time/File performance metric as compared to other Github projects.

Table IV-B shows the execution performance for smaller plugins obtained from WordPress plugins website. The LoC for each project is consistent based on both the project size and the total number of PHP files. In terms of processing time and inference time, Sportpress takes much longer with 419.34 seconds and 3.48 seconds respectively, even though having fewer LoC than Simple Jobs Board. However, it is worth noting that Sportpress has a higher number of PHP files, and this significantly affects the execution time as the evaluation is done based on the file-level granularity. Surprisingly, in terms of Time/LoC and Time/File metrics, smaller projects like Appointment Booking Calendar and Payment Form for PayPal Pro recorded higher values as compared to larger projects like Sportpress and Simple Jobs Board. As for the popular projects, this variance reflects the different code complexity and style across different projects.

We expect LoC and processing time to exhibit some linear dependence. To verify this, we visualise the plot of processing time against LoC for all software projects in Figure 3. The figure shows a consistent pattern between processing time and LoC across all the software projects. The scatter of the points in the plot follows a pattern, where the LoC increase, the processing time also increases. This relationship is further demonstrated through the regression lines added in the figure. We can see that Sportpress has a near-perfect linear trend throughout all the data points. However, several outliers can also be seen in the figure, especially the one that belongs to Symphony. This outlier has a value of 2326 LoC and 50.09 seconds of processing time. This specific data point is far from the projected trends of all the software projects. We inspected the file representing that data point, which is FrameworkExtension.php. This file contains a lot of nested if-else conditions in most functions, which explains the longer processing time. The creation of CFG for nested if-else conditions takes longer compared to simpler source code.

Overall, this performance analysis shows that DeepTective is an efficient model which can scale without problems to larger code bases. In the worst case scenario it takes less than half an hour to analyse a 27MB project. Considering that this kind of vulnerability detection is an offline task that is performed only periodically on a whole project, this cost is negligible.

B. Classification Performance

We established that DeepTective is usable in terms of execution performance. We now consider the usability in terms of vulnerability detection. As discussed above, we make the assumption that the popular projects currently do not have any security vulnerabilities. Hence we regard any positive reported by DeepTective as a false positive. In Table IV we report the accuracy of File-A and File-G for all projects. As observed in Section V-B, File-G has fewer FP than File-A, which translates to a higher accuracy on a vulnerability-free dataset. Hence we recommend to use File-G especially if the code base is large and the priority is to reduce false positives.

TABLE V: Novel vulnerabilities detection task

Software project	TN	FP-SQLi	FP-XSS	FP-OSCI
Booking calendar	9	7	0	0
Payment form paypalpro	7	1	5	0
Paypal for digital goods	3	4	0	0
Sportpress	123	31	102	0
Simple Jobs Board	171	4	23	0

C. *Vulnerability Detection*

Finally, after having observed a good level of performance on our datasets, we attempt to use DeepTective to discover new 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. We assume that the smaller plugins we considered may indeed contain vulnerabilities. Our priority is to minimise the manual effort spent reviewing reported positives. As machine learning techniques make no promise of completeness, it is preferable to miss some detections but focus the code reviewing efforts to hopefully identify some existing flaws.

We use File-G (our practical model with fewer false positives) to detect potentially vulnerable files from the smaller plugins projects. That yields 177 potentially vulnerable files across two vulnerabilities, SQLi and XSS. Table V shows the results for the novel vulnerabilities detection task.

In absence of ground truth, we need to resort to manual inspection to verify the results. Several appeared suspicious (say concatenate a SQL string to a variable) but we did not have sufficient familiarity with the application to determine unequivocally if they constituted a vulnerability. However, we were able to confirm 4 of these vulnerabilities as actual security vulnerabilities, and we responsibly disclosed our findings to the respective developers. We publicly disclosed 2 of them after they were patched and we describe them below.

CVE-2020-14092. We found a SQL injection vulnerability in the plugin "Payment Form for PayPal Pro". It allowed any user to perform any SQL query they wanted, including retrieving user login information. This received a CVSS score of 9.8 (critical). Figure 4 shows the vulnerable code snippet from the source codes.

CVE-2020-13892. We found an XSS vulnerability in the "SportsPress" plugin, which allowed authenticated users to add malicious JavaScript to the WordPress installation. This received a CVSS score of 5.4 (medium). Figure 5 shows the vulnerable code snippet from the source code.

We perform a comparison between DeepTective and other vulnerability detection tools in terms of discovering real-world vulnerabilities. We tested the tools from Section V-C on these projects to see if they could detect either of the above vulnerabilities, but none succeeded. This demonstrates the effectiveness of DeepTective and its applicability in real-world usage.

Fig. 4: SQLi vulnerability CVE-2020-14092

```
function cp_ppp_init_ds(){
    $query_result = cp_ppp_ds( $_REQUEST );
    $err = mysqli_error( $cpcff_db_connect);
    if( !is_null( mysqli_connect_error()))
         $err .= mysqli_connect_error();
     if( $_REQUEST[ 'cffaction' ] == test_db_query ){
          print_r( ( ( empty( $err ) ) ? $query_result:$err));
     }else{
          $result_obj = new stdClass;
          if( !empty( $err ) ){ $result_obj->error = $err;
          }else{
               $result_obj->data = $query_result;
          }
          print(json_encode($result_obj));
     }
}
```

Fig. 5: XSS vulnerability CVE-2020-13892

```
public function save(){
    parent::save();
    if ( isset( $_POST[ 'sportpress_events_teams_delimiter']))
    update_option( 'sportpress_event_teams_delimiter',
        $_POST['sportpress_event_teams_delimiter']);
}
```

D. *Limitations and Threats to Validity*

In the context of our experiment, we selected GRU and GCN as our main architecture for DeepTective. Our choice of architecture was biased by our approach to this problem from a graph semantics perspective and sequence-of-tokens perspective. However, we acknowledge that other approaches can be implemented other than the one proposed in DeepTective.

Our experiment involves several web applications that are written in PHP language. We are aware that some of our techniques are only applicable to PHP language and cannot be applied to different programming languages such as C++, Java, and Python. However, the general methodology of DeepTective, where we combine GRU and GCN to learn long term dependencies and code semantics, can be applied to other programming languages with the right tools and tokeniser.

Our experiments are limited only to the top four vulnerabilities in PHP applications. Therefore, even though our approach can be easily extended to consider more CWEs for the multiclass classification task, we do not know if the new task's performance will be the same or on par with the one reported in this work.

In our experiment, we use three datasets: SARD, NVD and Git. The fundamental issue with the SARD dataset is that it is rather simplistic and not representative of real-world code samples. Furthermore, there are many duplicates post-processing as a majority of the code is duplicated across samples with only one or two lines changing. We found that occasionally duplicates across classes were far from ideal. In our case, we removed all duplicates. The NVD and Git datasets posed a different issue stemming from collecting the code samples. When we have a vulnerable commit, we define a vulnerable function as one where a line has been changed in the commit. The issue is that some commits we saw would include refactoring or code unrelated to the vulnerability fix where the author has included other features.

This issue introduces a label noise problem in our dataset.

VII. 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.

References

[1] R. Rabheru, H. Hanif, and S. Maffeis, "Deeptective: Detection of php vulnerabilities using hybrid graph neural networks," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ser. SAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1687–1690. [Online]. Available: https://doi.org/10.1145/3412841.3442132

[2] MITRE, "Browse cve vulnerabilities by date," 2020. [Online]. Available: https://www.cvedetails.com/browse-by-date.php

[3] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S P'06)*, 2006, pp. 6 pp.–263.

[4] J. Dahse and J. Schwenk, "Rips-a static source code analyser for vulnerabilities in php scripts," in *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*, 2010.

[5] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[6] S. Son and V. Shmatikov, "Saferphp: Finding semantic vulnerabilities in php applications," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2166956.2166964

[7] I. Medeiros, N. F. Neves, and M. Correia, "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*, ser. WWW '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 63–74. [Online]. Available: https://doi.org/10.1145/2566486.2568024

[8] I. Medeiros, N. Neves, and M. Correia, "Equipping WAP with WEAPONS to Detect Vulnerabilities: Practical Experience Report," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 630–637.

[9] P. J. C. Nunes, J. Fonseca, and M. Vieira, "phpsafe: A security analysis tool for oop web application plugins," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015, pp. 299–306.

[10] J. Huang, Y. Li, J. Zhang, and R. Dai, "UChecker: Automatically Detecting PHP-Based Unrestricted File Upload Vulnerabilities," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 581–592.

[11] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[12] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, pp. 757–762.

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[14] Y. Fang, S. Han, C. Huang, and R. Wu, "Tap: A static analysis model for php vulnerabilities based on token and deep learning technology," *PLOS ONE*, vol. 14, no. 11, pp. 1–19, 11 2019. [Online]. Available: https://doi.org/10.1371/journal.pone.0225196

[15] A. Fidalgo, I. Medeiros, P. Antunes, and N. Neves, "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)*, 2020, pp. 465–476.

[16] N. Guo, X. Li, H. Yin, and Y. Gao, "Vulhunter: An automated vulnerability detection system based on deep learning and bytecode," in *Information and Communications Security*. Cham: Springer International Publishing, 2020, pp. 199–218.

[17] I. Medeiros, N. Neves, and M. Correia, "Dekant: A static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–11. [Online]. Available: https://doi.org/10.1145/2931037.2931041

[18] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[19] J. Kronjee, A. Hommersom, and H. Vranken, "Discovering software vulnerabilities using data-flow analysis and machine learning," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3230833.3230856

[20] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2021.

[21] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2019.

[22] D. Filaretti and S. Maffeis, "An executable formal semantics of PHP," in *ECOOP 2014 - Object-Oriented Programming - 28th European Conference*, ser. LNCS, vol. 8586. Springer, 2014, pp. 567–592.

[23] Stanisław Pitucha, "phply," 2016, [Accessed April 18, 2020]. [Online]. Available: https://github.com/viraptor/phply

[24] David Beazley, "ply," 2017, [Accessed April 18, 2020]. [Online]. Available: https://www.dabeaz.com/ply/

[25] scikit-learn, "LabelEncoder," 2011, [Accessed April 18, 2020]. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html

[26] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, 2017, pp. 334–349.

[27] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[28] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[29] M. Fey, "Pytorch geometric documentation," 2020, [Accessed September 26, 2020]. [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html

[30] National Institute of Standards and Technology, "SARD," 2020, [Accessed April 10, 2020]. [Online]. Available: https://samate.nist.gov/index.php/Software_Assurance_Reference_Dataset.html

[31] Bertrand C. Stivalet, "PHP Vulnerability Test Suite," 2015, [Accessed April 10, 2020]. [Online]. Available: https://samate.nist.gov/SARD/view.php?tsID=103

[32] ——, "PHP Vulnerability Test Suite Generator," 2015, [Accessed April 10, 2020]. [Online]. Available: https://github.com/stivalet/PHP-Vuln-test-suite-generator

[33] National Institute of Standards and Technology, "National Vulnerability Database," 2020, [Accessed April 10, 2020]. [Online]. Available: https://nvd.nist.gov/

[34] DesignSecurity, "progpilot," 2017, [Accessed June 05, 2020]. [Online]. Available: https://github.com/designsecurity/progpilot

[35] V. Khliupko, *Composer*. Berkeley, CA: Apress, 2017, pp. 43–50. [Online]. Available: https://doi.org/10.1007/978-1-4842-2460-1_6