Classifying PHP Web Shells

Wilson Redd, Kaiden Nunes, Clayton Kingsbury, Blake Moss
CS 478, Winter 2019
Department of Computer Science
Brigham Young University

Abstract

Cybersecurity analysts are often faced with the daunting task of deciding whether files, emails, and other IT entities within their enterprise are malicious or benign. One of the well known attack techniques against organizations who host web servers includes uploading malicious files known as web shells. Web shells are programs that are written for a specific purpose in web scripting languages...[and] provide a means to communicate with the servers operating system via the interpreter of the web scripting languages [1]. PHP is a commonly targeted language since due to its ubiquitous nature in the web development community. Classification of PHP web shells through various machine learning techniques has shown great promise in detecting malicious files. This research focuses on applying many machine learning algorithms to this task. The preliminary results show that using machine learning models to detect malicious files is an effective approach to automating this process and that cybersecurity professionals, in conjunction with data scientists, should explore this area further.

1 Introduction

A February 2018 joint report by McAfee and the Center for Strategic and International Studies estimated that the economic cost of cybercrime totals just under 600 billion dollars [2]. Unfortunately, the true impacts of cybercrime are much deeper than just pocketbooks, with privacy, reputation, and identity all under attack in the digital age. More than ever before, adversaries in the cyber field are continually generating new techniques to evade preventative and detective methods. This introduces a challenging cat-and-mouse game where system defenders identify (usually via signature-based heuristics) and remediate (e.g., blocking an IP address or blacklisting a URL) only to find a modified threat already infecting their environment. Due to the rigid nature of classical signing techniques, the system defender must begin the cycle all over again. Classification tasks using supervised machine learning algorithms seek to find patterns of various feature values and pairings to accurately predict the probability of a particular class output. This methodology would seem to be apposite for the morphing nature of cyber threats, particularly malware, where many implementations are variants of the same design.

A popular form of malware is called a web shell. A web shell is a backdoor program designed to give remote access to an operating system via the web server's code interpreter. Although sometimes used for legitimate remote access purposes, the vast majority of web shells are designed to provide an attacker persistence on a targeted system. This allows for repeated and sustained access to internal features of operating and file systems. Sophisticated web shells even have advanced features to provide disruption (e.g., file removal, spamming abilities) and further exploitation (e.g., network scanners, credential dumpers, etc).

PHP, or Hypertext Preprocessor, is one of the most common interpreted languages in web development and as such is widely used in many popular web technologies. One study found that PHP code accounts for around 80% of the code on the most common websites on the internet [5]. Due to the strong usage rate on web servers and wide community support, web shells are commonly written in PHP. Our research focused on designing an accurate classifier that could reliably predict whether a file of any type is a PHP web shell or not. This work involved elements of data gathering, feature selection and extraction, model selection, and statistical verification which are detailed below.

2 Methods

2.1 Data Sources

The initial data sources for this project include a web shell repository created by security researchers at Black Arch Linux (`https://github.com/BlackArch/webshells`) and generated web shells via the backdoor post-exploitation tool, weevely (`https://github.com/epinna/weevely3`). Web shells ranged from complex and obfuscated, to simple and trivial to interpret. Non-web shell samples were gathered from popular PHP frameworks such as Laravel (`https://github.com/laravel/laravel`) and Symfony (`https://github.com/symfony/symfony`), as well as the very popular PHPmailer project (`https://github.com/PHPMailer/PHPMailer`). These code bases are extremely common in enterprise web

development projects and as such make an apt inclusion in the dataset. Our initial dataset consisted of the following:

Web Shell Instances	Non-Web Shell Instances	Total Instances
1062	1497	2559

2.2 Initial Feature Selection

A popular tool for detecting malicious web shells, particularly PHP, is the NeoPI project (`https://github.com/Neohapsis/NeoPI`). NeoPI runs static, statistical analysis on a file to determine probability of maliciousness. For our first feature extraction process, it was determined that the seven tests performed by NeoPI would be suitable features for initial modeling and testing. The tests performed by NeoPI (and so named by the code author) and corresponding sections of the feature set are:

1. **Incidence of Coincidence**: Indicates the distribution of characters in a given text which provides better insight into both plaintext and ciphertext and helps analysts detect language distribution of the text/ciphertext to determine if a file is potentially obfuscated, encrypted or written by foreign adversaries.
2. **Entropy**: Shannon Entropy can be a helpful indicator of encryption or obfuscation, potentially hinting at a malicious file.
3. **Nasty Signature Count**: Uses a regular expression parser to determine the number of matches a file has with patterns common to web shells and other back doors. These patterns include using function calls for base64 encoding, exec functionality, process/shell execution commands and other indicator patterns.
4. **Super Nasty Signature Count**: Again, utilizes a regular expression parser to look for patterns considered especially indicative of malicious files. These patterns include the use of global variables that are populated by requests to the web page.
5. **Eval Uses**: Uses of the `eval` command, are rarely used in properly coded PHP and represent a significant security risk since they interpret strings as executable code. A parser analyzes the amount of times eval is used in a file.
6. **Compression Ratio**: The compression ratio compares the length of the raw data to a compressed version of the data. Obviously, if the data is already compressed the ratio will be around 1. This can help indicate whether compression was used which is a potential indicator of malware.
7. **Longest word Length**: Some web shells use extremely long strings (especially when invoked by an `eval` command) to execute malicious operations.

The NeoPI analysis script was integrated into a feature extraction tool that reported features on the web shell and non-web shell raw file datasets.

2.3 Dataset Augmentation, Feature Improvements and Feature Reduction

Although initial results using this dataset were encouraging, there were concerns that the dataset was not representative of: 1) typical file varieties of most web servers (i.e., a mix of PHP, JavaScript, HTML, images, and other file formats) and 2) all the various webshell implementations due to our limited dataset. In order to design a better generalizing classifier, we received permission from the Church Educational System (CES) security team at Brigham Young University to extract features from a partially labeled dataset in their possession. This increased our dataset to the following:

Web Shell Instances	Non-Web Shell Instances	Total Instances
1185	9314	10499

The addition of the CES dataset provided a more realistic representation of data varieties and presented a more robust training set for our classifier. However, it should be noted that along with this augmentation came a stark imbalance of web shell instances to non-web shell instances. This represents a common issue in machine learning applications in cybersecurity. On average, there are relatively few malicious samples compared to benign. This is somewhat intuitive, as malicious files, though not inherently illegal, are not usually distributed through publicly accessible mediums. Furthermore, the vast majority of software developers are not employed to write malicious code, thereby producing the imbalance represented in the dataset.

In addition to dataset augmentation, more features were gathered to further enhance PHP-specific web shell detection. These indicators were extracted using a set of YARA rules written by the PHP-Malware-Finder open-source project (`https://github.com/nbs-system/php-malware-finder`). YARA is a popular malware textual and binary analyzer that matches text and binary features of a file with user-defined rules. The rules from the above-mentioned project were tailored to specific PHP web shell signatures and patterns, making it a cogent complement to the existing features already extracted via NeoPI. It was decided that that new features would be derived for the number of matching occurrences of each major YARA rule. An explanation of those rules with its original name (as given by the code authors) are as follows:

1. **Non-printable characters**: Characters that are non-printable fall in a range of Hex values. This rule looks for a specific pattern of non-printable characters.
2. **Obfuscated**: Obfuscation is a commonly employed tactic to confuse both human and machine analyzers. This rule identifies patterns of obfuscation (i.e., function calls, shell-specific obfuscation expressions) in a file.
3. **Password Protected**: Masking detection by requiring authentication to gain access to the web shell functionality is also a common method undertaken by attackers. This rule looks for patterns that might indicate the file has a password protected section.

4. **Embedded Images**: Embedding web shell code in a benign looking image (e.g., PNG, JPG, etc.) also confuses human and machine analysts. This rule detects images that have embedded PHP that matches other suspicious rules.

5. **Dangerous PHP Code**: There are a number of PHP functions that are uncommon in benign PHP files and very common in web shells. These functions usually have to do with system level APIs (e.g., opening command line shells, creating processes, opening sockets, etc.). This rule searches for calls matching this list of dangerous function names.

6. **Dodgy PHP Code**: Different settings and keyword/binary patterns can make PHP code more easily weaponized (e.g., disabling magic quotes, regex replace-and-execute, etc.). This rule looks for occurrences of such settings and patterns.

7. **Suspicious Encoding**: Although there are some legitimate uses of encoding in PHP, it is also commonly used as an obfuscation method. Combined with hex encoding of dangerous PHP functions, encoding can be a helpful indicator of a web shell. This rule searches for various encoding methods.

8. **Dodgy Strings**: Often in a web shell, certain strings indicate a high probability of the presence of a web shell or other malicious activity. These strings include various system-level executable names, file locations and keywords (e.g., /etc/passwd, cmd.exe, nc -l), popular reconnaissance, password cracking, and exploitation tool names and commands (meterpreter, hashcrack, exploit), as well as "hacking" colloquiums and popular web shell names (sun-tzu, c99, b374k). This rule parses for these and similar strings.

9. **Cloudflare Bypass**: Cloudfare is a popular intermediary service designed to give website protection by providing an inspecting proxy to handle requests from users to a website. This allows request inspection to determine whether a user request is legitimate or not. This rule searches for evidence that Cloudflare tampering is occurring. However this rule is not robust and thus does not constitute a valuable feature.

10. **Websites**: References to websites, although not malicious by themselves, are often used to retrieve further exploitation tools or communicate with a command and control server. Many hard-coded URLS are indicators that the file was written by someone with malicious intents. Those URLs are scanned for with this rule.

In addition to increasing both dataset instances and feature set, principle component analysis (PCA) was attempted in order to reduce feature set size while retaining as much information as possible. Using Sklearn's PCA method, the dataset was reduced to four principle components. However, in practice it could be difficult to justify performing PCA on every instance since this classifier is envisioned to be an online tool for analysts to use and time costs also could be too great.

3 Models

Early in the research process, it was important to understand preliminary performances on several different learning models. Using implementations from PyTorch [3], Sklearn[4], and CS 478 class projects, accuracy measurements on the initial dataset (2559 instances) revealed clear performance separation for the following classifier models: Multi-Layer Perceptron (**MLP**), Gradient Boosting Tree Model (**GBM**), K-Nearest Neighbors (**KNN**), Decision Tree (**DT**), Scaled Gamma Support Vector Machine (**SVM**), and Multinomial Naive Bayes (**MNB**). The below table shows a summary of those results:

Model	Averaged Accuracy
Baseline	.585
MLP	.941
GBM	**.984**
KNN	.903
DT	.680
SVM	.703
MNB	.702

Using these results, it was decided to focus on the models that showed the most promise or, in the case of DT, could be improved by an efficient ensemble: MLP, GBM, KNN, and DT (subsequently replaced by Random Forest). A brief explanation of each model and improvements that were made are detailed below:

3.1 MLP with Backpropagation

The Multi-layer Perceptron algorithm is an algorithm that can be easily adapted to fit many complex tasks. We were hopeful that we could easily improve this model by varying the number of hidden nodes and iterating over the different hyper parameters.

For our initial tests, we decided to implement a simple, single layer network in PyTorch to understand the baseline complexity of our problem. We decided to start with twice the number of hidden nodes as there were input features. For the initial data set there were 14 hidden nodes and for the expanded data set there were 30 hidden nodes. For our tests, we used a learning rate of 0.01, momentum of 0.9, stochastic gradient descent as our optimizer, cross entropy loss as our loss metric, and rectified linear units (relu) as our activation function. We ran five tests with a random 70/30 train/test split and averaged the accuracy, recall, precision, and F1 score over the five tests. The results were 94% accuracy on the initial dataset and 93% accuracy on the test dataset.

To improve the performance of our MLP, we decided adjust the number of hidden nodes as well as the depth of our MLP to understand how that would improve our F1 score/accuracy. We began by keeping the network single-layered and just increased the number of hidden nodes. We doubled the number of hidden nodes each test to determine the optimal number of nodes that would produce maximal accuracy. For the initial dataset, we found that adding more hidden nodes to our network did not significantly improve our accuracy. For the

extended dataset, we found that 60 hidden nodes improved the accuracy from 93% to 94%, a 16% increase.

To improve our model further, we modified the depth of our MLP. With the optimal hidden nodes from the tests above we added layers one by one to see how that impacted the accuracy/F1 score. For the initial dataset, we found that having 3 layers improved the accuracy from 94% to 97%, a 50% increase. For the extended dataset, we found that 4 layers improved the accuracy from 94% to 97%, a 57% increase.

Finally, we decided to experiment with different optimizers PyTorch has available to discover if that would allow the model to converge faster. Specifically, we tried the Adadelta and Adam optimizers. For the initial dataset, we found that Adadelta and Adam only improved the accuracy for the 3 layer SGD model by 0.5%. However, on the extended data set, Adadelta improved the accuracy for the 4 layer model by 1% and Adam improved the accuracy by 2% resulting in the best accuracy of 99%.

As we iteratively, adjusted the model architecture and experimented with different PyTorch optimizers we were able to improve the model from 94% to 97% accuracy on our initial dataset and from 93% to 99% on our extended dataset.

3.2 K-Nearest Neighbor (KNN)

In testing this model, we experimented with modifying the number of neighbors, as well as using Euclidean, Manhattan, and no distance weighting. We normalized all feature input, used a 75/25 random training/test split, and averaged 10 separate runs to find the accuracy, recall, precision, and F1 score of the model.

Our first attempt used our in-house implementation of KNN. However, it soon became apparent that our model was inadequate for dealing with continuous input features, as it always predicted that the web shell was benign. We then used sci-kit's implementation of KNN, using k = 5 and no distance weighting, which produced an accuracy of 90.329% for the initial dataset and 93.876% for the extended dataset.

By experimenting with distance weighting and the value of k, we managed to increase the accuracy of the initial dataset to 93.975%, when k = 11 and using Euclidean distance weighting. Using these same hyperparameters, our extended dataset's accuracy increased to 96.452%.

3.3 Decision Tree and Random Forest

Initially, our testing on a decision tree using 10 cross-validation resulted in poor accuracy. The accuracy for the decision tree resulted in was 68%, a lot of room for improvement.. For the decision tree, we figured over-fitting was one of the main issues, so we decided that a random forest would be the best method to improve our results because random forest trains many decision trees with different subsets of the original data set.

A random forest is an ensemble learner based on decision trees, that creates multiple decision trees from a random subset of the data and uses the result from the each of the trees to vote for the prediction. Implementing the tree was simple because it was just creating a class that randomly generates subset of the data and then trains a tree on that subset. We used the instances that were not used in the decision tree to create a validation set which was used to weight the votes. This gave the greatest weight to the most accurate tree. We found that the number of trees had little effect on the data set as long as the number was greater than five trees. With our implementation of a random forest, we increased the accuracy of 10-fold CV to 72%. The accuracy was low because our decision tree model did not have a good method to bin continuous attributes. We implemented multiple binning methods to test out if that would increase the accuracy. Our binning methods included binary splitting based on the mean of the attributes, binary splitting based on the best information gain metrics based on a threshold, and splitting based on one of the fore mentioned methods based on the greatest variance on an attribute. Another method that we implemented was training clusters based on individual attributes using the k-means model, which gave the best results. Even with training clusters with a range of k values(2-10) the model would still choose a binary split because these splits had the best silhouette scores. We found our binning methods were not effective at significantly increasing the accuracy of our model.

To get our final results, we ran the dataset on the Sklearn's random forest model. Along with having a superior binning method, Sklearn's random forest model uses Gini Impurity to decide where to split instead of standard information gain, resulting in better accuracy for continuous values. With this random forest, our results for accuracy were 98%, much better than our the initial results with our decision tree and random forest implementations.

3.4 Gradient Boosting Classifier

Gradient boosting is an ensemble learning method commonly used for both regression and classification tasks. The principle idea of gradient boosting relies on the observation that a weak learner F_m can be made iteratively stronger by adding an estimator $h(x)$ that corrects the errors of the model. This forms a new model $F_{m+1}(x)$ where ideally $F_{m+1}(x) = F_m(x) + h(x) = y$. A decision tree is usually used to estimate the best $h(x)$ that predicts correct error correction values. This method can be iteratively applied so as to create a robust ensemble model.

Sklearn's Gradient Boosting Classifier implementation was used for this research due to its already well-optimized performance. Accuracy on the initial dataset was relatively high (at 98%). This was encouraging but further improvement was desired. Since the Sklearn's GBC is a decision tree ensemble, there were several hyperparameters that were available to be tuned. Minimum samples required for a split, number of trees involved in the ensemble, maximum number of features considered for splitting, as well as the maximum depth of each tree were decided as the tunable hyperparameters. Through a 10-fold cross-validation (CV) grid search, optimal parameters were found. This resulted in a slight increase in all of the performance measurements used. A 10-fold CV averaged accuracy metric was used to calculate accuracy performance for this model.

4 Performance Metrics

In typical classification tasks, accuracy is usually regarded as a apt indicator of a performant model. Indeed, accuracy was

the deciding factor utilized to determine which models to focus on. However, for particular tasks, measurements such as recall, precision, and the F1 score give more insight into performance. Recall can be defined as the ratio: $\frac{TP}{TP+FN}$, where TP is the frequency of true-positives and FN is the frequency of false-negatives. This statistic is particularly important in web shell detection performance because of the high cost of false negatives. Human analysts will be alerted to and can usually determine false-positives quickly but if a file is deemed benign when it is actually not, there is a real risk of the malicious file not being detected. Precision is similar to recall: $\frac{TP}{TP+FP}$, where FP is frequency of false-positives. Although not as critical of a measurement as recall, precision helps to understand a model's tendency to classify non-malicious files as web shells. This tendency can be tedious for human analysts who must react to the alerts of the classifier. Finally, the F1 score is a weighted mix of precision and recall: $2x\frac{Precision*Recall}{Precision+Recall}$. In comparison with accuracy, F1 performance more heavily considers false-positive and false-negative rates. This score is of more value for a web shell classifier because the most impactful costs of cybersecurity prediction are false-negatives, followed by false positives. Because of this rationale, these three additional statistics, along with accuracy were reported for the final model/dataset iteration.

5 Comparison of Selected Models

5.1 Initial Results

Using the metrics described in the previous section, the first iterations of our selected models were tested on the initial dataset. The results follow:

Results on Initial Dataset (2559 instances)

Model	Accuracy	Recall	Precision	F1
Baseline	.585	-	-	-
MLP	.941	.939	.941	.940
GBM	**.984**	**.972**	**.984**	**.978**
KNN	.903	.896	.903	.899
RF	.718	.236	.957	.4731

From these results, it can be observed that MLP, GBM, and KNN perform fairly well. GBM scores highest in all the performance metrics, indicating that this model could generalize well to the expanded dataset. Our random forest implementation was substantially lower. That low score however, can most likely be attributed to using a self-implementation instead of a commodity version.

5.2 Final Results

Using the iterative approaches described in sections 2 and 3 to improve the dataset and models respectively, the results for each model on the improved dataset are shown below, along with the performance metrics discussed in section 4:

Final Results on Augmented and Expanded Dataset (10499 instances)

Model	Accuracy	Recall	Precision	F1
Baseline	.887	-	-	-
MLP	.990	**.972**	.976	**.974**
GBM	.990	.957	.981	.967
KNN	.965	.890	.928	.908
RF	**.994**	.958	**.985**	.971

As the results show, performance was high among all the classifiers. Sklearn's Random Forest implementation along with PyTorch's MLP were the most performant. An interesting observation to be made is that although RF has a slight edge over MLP in terms of accuracy, MLP's recall score is more than .02 higher. Referencing the discussion in section 4, it would appear that MLP, with a higher recall and F1 score, is more suited to minimize principle cybersecurity risks like false-negatives.

Surprisingly, performance on the PCA-reduced dataset was even higher than the the results above. For MLP, RF, GBM, and KNN models, perfect or near-perfect metrics were reported. This was interesting and could be attributed to the principal components having a better structural representation of the maliciousness of a file with less features, thus allowing the models to be less prone to mistakes caused by misleading relationships between features of the full dataset. However, further research needs to be conducted in order to truly validate the PCA process as well as the model results.

To continue iterating on the current results, an open-source repository has been established at `https://github.com/bmoss6/ml_478_php_webshell`. This contains code implementations for the final models used, as well as dataset and feature extraction information.

6 Conclusion

6.1 Concluding Observations

Important observations relevant to designing efficient machine learning solutions to cybersecurity problems became apparent throughout the course of this research. The following observations were deemed as noteworthy:

Importance of Industry Expertise
For any non-trivial machine learning solution, industry experience is paramount in both data gathering and feature selection. The cybersecurity field is no exception. Attribution of the high performance of the various classifiers could be related to the reliance of industry expertise in the feature selection process. Both tools (NeoPI and the custom YARA rules) represent significant knowledge expertise specific to PHP web shells. The features derived from these tools most likely accounted for a large part of the classifiers' successes.

Limitations for Evolving and Advanced Threats
The premise of supervised machine learning approaches, like this research, rely on unseen instances of classification data being similar to instances the model has been trained to recognize. If instances evolve to the point of bearing no resemblance to trained data, then empirical results will be poor. Although limiting the scope of this project to PHP-specific web

shells helped reduce the potential for such an evolution (since PHP is a fairly stable language with reliable historical patterns), the cyclic nature of cybercrime and security could still introduce novel attack vectors, even within the confines of PHP code.

Furthermore, it is highly probable that due to the public nature of our dataset, web shells produced by cutting-edge and well-sponsored groups are not represented. Cybercrime organizations at the top of the sophistication hierarchy most likely do not reveal active exploitation code purposefully and furthermore are far more advanced in method than commodity web shells like those in our dataset. Because of this, it is difficult to provide an effective machine learning solution that combats such threats.

Empty Features

During feature extraction, a small number of features that measured the frequency of particular PHP expressions did not appear the vast majority, and sometimes all, of the data instances gathered. Those features included the "Super Nasty Signiature Count" from NeoPI and "CloudFlare Bypass", and "Suspicious Encoding" from the YARA rules. Although these features did not appear in our specific dataset, removing features from continued iterations of the project would be unwise, since these features have high correlation to web shell detection. This also illustrates the limitation of our dataset as currently constructed.

Complexity of Explanation

Particularly in neural networks, explaining the causal relationships between feature vector and classification decision can be difficult. Due to the inherent complexities involved in neural network architecture, security professionals can often be left with a classification but no substantial explanation for why the classification was made. Even though the neural network performed exceptionally on the dataset, it can be difficult to mandate policy decisions (e.g., blocking a file, blacklisting a user, etc.) without a logical explanation. Since cybersecurity findings often require proactive or reactive changes to critical systems, challenges can arise when persuading organization leadership to make changes based on neural network results alone.

6.2 Further Research

It is anticipated that research will be conducted into developing a usable tool for security analysts, as well as improving the dataset, model(s) and feature selection process.

Analyst Tool and Data Expansion

In order to make this research accessible to the security community, an open source command line tool and accompanying python library is being developed. This tool will allow security analysts to scan a list of files and get classification results for PHP web shells. Analysts will also be able to improve the model by manually classifying data instances whose probability output by the model fall within a confidence range (i.e. 40-80%). These newly labeled datasets will become part of the training set for the model. This will allow for the model to adjust to new threat vectors as quickly as possible.

Feature Improvement

There exists a plethora of valuable static and dynamic code analysis tests that could be useful to incorporate as new features for this research. Analytical engines like Cuckoo sandboxes and additional YARA rules could provide better insight into patterns of web shells. It is expected that industry expertise will significantly contribute to further development of the feature extraction processes.

Expansion to Other Languages

As prevalent as PHP is in the web development world, there are numerous other languages that web shells utilize for nefarious purposes. It would be important to broaden the scope of future research to include shells for languages such as ASP, Javascript, Python, and other commonly used languages found on web servers.

Comparison against Currently Used Tools

Finally, to truly develop a standard of efficacy, the classifier will need to be compared with both open-source and commercial tools such as anti-virus scanners, malware scanners, and other machine learning solutions aimed at detecting malware, in particular web shells. Empirically comparing these tools will allow for scrutinization as to the benefit of machine learning based solutions versus traditional threat hunting standards.

Acknowledgments

Appreciation is expressed to Black Arch Linux, the Weevely project, Laravel, Symfony, and PHPmailer for providing a large portion of the dataset used. Additionally, sincere thanks to both the developers of NeoPI and PHP-Malware-Finder for providing the bulk of feature-extraction code utilized in this project. Without the security-specific knowledge built into those projects, this classification task would have proved much more difficult. Also, recognition to the CES security team for providing additional data that allowed further improvements on the dataset and ultimately a more robust classifier.

References

[1] Jinsuk Kim et al. "WebSHArk 1.0: A Benchmark Collection for Malicious Web Shell Detection". In: *JIPS* 11 (2015), pp. 229–238.

[2] James Lewis. *Economic Impact of Cybercrime - No Slowing Down*. URL: https://www.mcafee.com/enterprise/en-us/assets/reports/restricted/rp-economic-impact-cybercrime.pdf.

[3] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017.

[4] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[5] W3 Techs. *Usage statistics and market share of PHP for websites*. URL: https://w3techs.com/technologies/details/pl-php/all/all.