A Modular and Model-Agnostic LLM Framework for PHP Webshell Detection

Bibek Neupane
Department of Computer Science
William Paterson University
Wayne, NJ, USA
neupaneb1@student.wpunj.edu

Cyril S. Ku
Department of Computer Science
William Paterson University
Wayne, NJ, USA
kuc@wpunj.edu

Kiho Lim
Department of Computer Science
William Paterson University
Wayne, NJ, USA
limk2@wpunj.edu

***Abstract*—PHP currently powers a large portion of modern web applications, which makes PHP-based applications a frequent target for attackers. A PHP webshell is a malicious backdoor deployed on servers running PHP applications and is often obfuscated, embedded within otherwise normal-looking code, or designed with multi-step execution logic, making it difficult for traditional methods such as signature-based or regex-based detection to identify. This study proposes a modular, locally deployable, and model-agnostic framework that performs context-aware analysis of PHP files using open-source large language models. Experimental results show that the proposed framework provides an effective, privacy-preserving, and adaptable solution for PHP webshell detection in real-world environments.**

***Index Terms*—PHP webshell, PHP malware, backdoor detection, large language models, malware detection**

I. Introduction

PHP has long been a foundational technology for modern web applications and continues to power a large portion of the web today [1]. Modern Content Management Systems (CMSs) such as WordPress and Joomla, frameworks such as Laravel, and many custom application dashboards are written in PHP. However, the widespread use of PHP also introduces significant security risks, including file-upload, remote-code-execution, and command-injection vulnerabilities. The exploitation of such vulnerabilities on a PHP server can allow a threat actor to upload a backdoor for post-exploitation and persistence, commonly referred to as a PHP webshell.

PHP webshells may exist as standalone files or be embedded within legitimate code. They can provide attackers with remote command execution, persistence, and other post-exploitation capabilities through a web interface or back-connection mechanisms. Webshells range from simple one-line payloads to more extensive malicious applications that support activities such as mass defacement, cryptocurrency mining, and database dumping. In severe cases, they may enable attackers to compromise the backend server, the underlying operating system, and associated databases, and in some situations, pivot further into the local network.

Webshells remain one of the major tools used to compromise web servers. In fact, about 49.21% of compromised websites were found to contain at least one backdoor at the point of infection [2]. Supply-chain attacks involving widely used open-source libraries and plugins, such as the PHPMailer command injection vulnerability (CVE-2016-10033) [3] and the PHP 8.1.0-dev User-Agentt backdoor remote code execution incident [4], continue to expose web servers to threat actors. These incidents illustrate how malicious code can be introduced into widely used software components and affect users at scale. The stealthy and easy-to-blend nature of webshells makes them particularly difficult to detect.

Currently, several approaches are used for webshell detection, including signature-based and regex-based detection, YARA rules, cloud-based scanners, and proprietary APIs. Most organizations rely on one or a combination of these methods. However, these approaches often struggle to detect malware that blends into existing codebases and employs techniques such as obfuscation, encoding, whitespace manipulation, and function indirection.

Modern, dynamic, and multi-stage payloads require deeper contextual reasoning, which makes detection more challenging. Although traditional signature-based and regex-based methods are fast and lightweight, they can be bypassed through small structural modifications and may generate false positives while still missing obfuscated or previously unseen threats. Cloud-based and API-driven solutions also introduce privacy and compliance concerns. In addition, manual code review becomes impractical when dealing with large and frequently changing codebases. These challenges highlight the need for a flexible, privacy-preserving, and open-source solution that can analyze context and adapt to diverse coding patterns.

To address these challenges, we propose a privacy-preserving, modular, and model-agnostic framework for PHP webshell detection based on open-source large language models. The framework enables context-aware analysis of raw PHP code, supports local deployment in privacy-sensitive environments, and allows flexible integration of different open-source models without major architectural changes.

The remainder of this paper is organized as follows. Section II reviews related work. Section III presents the proposed framework. Section IV describes the experimental setup and evaluation results. Section V discusses the strengths and limitations of the framework, and Section VI concludes the paper.

979-8-3195-3238-1/26/$31.00 ©2026 IEEE
DOI 10.1109/IDS69480.2026.00010

II. Background and Related Work

A. Traditional PHP Malware Detection Methods

Most traditional approaches for PHP malware detection rely on static rules or pattern matching. Regex-based scanning, for example, uses predefined patterns to identify suspicious PHP functions in source code. These often include functions such as `eval()` and `system()`, which can enable direct code execution from user-controlled input. One example is the open-source tool Worldfind [5], which uses a large collection of regular expressions to flag suspicious patterns in PHP files.

Predefined YARA rules are also used for the detection of PHP webshells. The U.S. National Security Agency (NSA) provides a repository of YARA rules and other static analysis tools for detecting PHP webshells [6], [7].

Signature-based detection approaches have also been used. For example, the PHP Shell Detector project by *emposha* [8] uses a local and remote database of known webshell signatures for detection.

To monitor unauthorized modifications in codebases, techniques such as file integrity monitoring are also used. The NSA's *Detect and Prevent Web Shell Malware* guidance [7] recommends implementing file integrity monitoring to block or alert on unexpected file changes.

Although these techniques are fast and lightweight, they share important limitations. Techniques such as code obfuscation, encoding, and insertion of unrelated code can alter a file's signature or pattern, allowing malware to evade traditional detection mechanisms [9]. In addition, webshells that do not match signatures in known databases may bypass these approaches entirely.

B. Machine-Learning-Based Approaches

Machine-learning-based and deep-learning-based approaches have been introduced to address the limitations of traditional methods.

Nguyen et al. [10] proposed a model that transforms PHP source code into opcode sequences and applies a Convolutional Neural Network (CNN) for classification, achieving 99.02% accuracy on a dataset filtered using YARA rules. *Tian et al.* [11] proposed an approach that represents each token from an HTTP request using Word2Vec, arranges these vectors into a fixed-size matrix, and then applies a CNN to classify the request as malicious or benign. Their approach showed better performance than traditional keyword-based detection methods.

Zhou et al. [12] compiled PHP files into opcode sequences and applied an LSTM-based deep learning model. They found that a single-layer LSTM achieved over 95% accuracy, while deeper multi-layer models performed worse because of overfitting and gradient-related issues. *Gogoi et al.* [13] analyzed function calls and the use of PHP superglobal variables through static AST analysis and applied an LSTM model for classification, achieving a detection accuracy of approximately 97%. *Fang et al.* [14] compared TextCNN and Transformer architectures for webshell detection using both raw PHP source code and AST representations and found that the Transformer model performed better across both feature types.

Li et al. [15] presented ShellBreaker, which correlates syntactic and semantic features from PHP code, such as communication behavior, run-time environment adaptation, and sensitive operations, and applies classical machine-learning techniques to detect webshells, achieving a detection rate of 91.7%.

Recently, *Bai and Zhu* [16] proposed WebshellHunter, which applies BERT-based AST vectorization followed by a CNN–BiLSTM model to enhance feature extraction from obfuscated PHP code and reported improved detection accuracy.

Although machine-learning-based approaches have shown strong performance, their effectiveness depends heavily on the type and quality of the dataset and the feature representations used, such as AST structures or opcode sequences. Some models exhibit overfitting in deeper architectures, while others rely on heavily curated datasets that may not adequately reflect real-world production PHP code. To address these limitations, more adaptable and context-aware detection methods are still needed.

C. Closed-Source API-Based Detection

Closed-source large language models and cloud-based APIs can also be applied to PHP webshell detection. Proprietary models such as Google's Gemini 1.5 Flash, as well as platforms such as VirusTotal API and Cloudflare Security Center, may be used to analyze PHP files for malicious behavior.

However, these proprietary and cloud-based solutions typically require source code to be uploaded to external servers, which raises privacy and compliance concerns. When applied to large codebases, they may also become less practical because of usage limits, rate limits, and higher operational costs. In addition, such services may introduce uncertainty regarding how submitted code is stored, processed, or retained, which can be a significant concern when the scanned codebase contains sensitive information.

III. Proposed Method

To address the limitations of the previously mentioned studies and methods, we propose a privacy-focused, modular, and model-agnostic PHP detection framework. The framework is based on open-source large language models to provide context-aware analysis of PHP files.

A. System Architecture Overview

Instead of relying on predefined rules, handcrafted features, or known signatures, the proposed modular framework uses open-source large language models as the analysis engine to classify raw PHP code. To enable more effective context-aware evaluation of the code, the framework uses coder models, as these models are extensively pre-trained on source code and generally exhibit stronger code understanding capabilities than general-purpose reasoning models.

Algorithm 1 PHP Webshell Detection Workflow

Require: PHP files F, LLM model M
Ensure: File-level classification results

```
for each file f ∈ F do
    Preprocess f
    Split f into chunks {c_1, ..., c_n}
    for each chunk c_i do
        Construct prompt and run LLM inference
    end for
    Aggregate chunk results to obtain file label
end for
```

As shown in Fig. 1, the overall process is organized into sequential modules that handle file preprocessing and context-aware chunking, prompt construction, model-agnostic LLM-based inference, and final result aggregation.

Algorithm 1 summarizes the overall workflow of the proposed framework.

B. Detection Framework Components

The proposed framework consists of five independent modules. The first is the preprocessing module, which takes raw PHP files as input, validates the file type, normalizes character encodings, and performs deduplication by hashing file contents. This ensures that only valid, readable, and unique PHP files are passed to the detection pipeline. By filtering non-PHP content at this initial stage, the framework significantly reduces the computational load and resource consumption of the analysis engine.

The second module is the chunking module. To ensure that each file is properly analyzed, the input passed to the analysis engine must remain within the model's maximum context length. This is handled by the chunking module, which extracts the maximum context length of the local model from its configuration files. This dynamic extraction allows the framework to adapt to different models without manual configuration. If a file exceeds the maximum context length, it is divided into smaller chunks so that the combined size of the prompt and the chunk remains within the limit. A sliding-window overlap is used to partially preserve code context even after the file is split into multiple chunks.

The third module is the prompt construction module. It combines each chunk with a system instruction, evaluation rules, and strict output formatting to construct a logical prompt. The resulting prompt is then processed by the model's tokenizer using the chat template defined in the model's configuration files. This ensures that each prompt follows the input structure expected by the model. As a result, the classification process remains deterministic, model-agnostic, and consistent across different models.

The fourth module is the LLM analysis engine, which can incorporate any locally deployed open-source LLM, preferably a model with strong code understanding and reasoning capabilities. Since the framework is designed to be model-agnostic, different LLMs such as Qwen Coder, DeepSeek Coder, StarCoder, or CodeLlama can be integrated without major modifications to the other modules. This module performs binary classification for each chunk by labeling it as either benign or malicious and also generates a short explanation for each prediction.

The output aggregation module is the final module of the framework. It aggregates the classification results across all chunks of a file using the rule that if any chunk is classified as malicious, the parent file is labeled as malicious. This aggregation strategy prioritizes recall and ensures that even if a webshell is embedded within an otherwise normal file, it is still detected. The final output consists of a binary label for each file, along with the technical reasoning behind the classification.

The final classification results can be exported in formats such as JSON or CSV, enabling seamless integration with incident response workflows. This modular separation of tasks also facilitates optimization and replacement of individual components without affecting the overall framework.

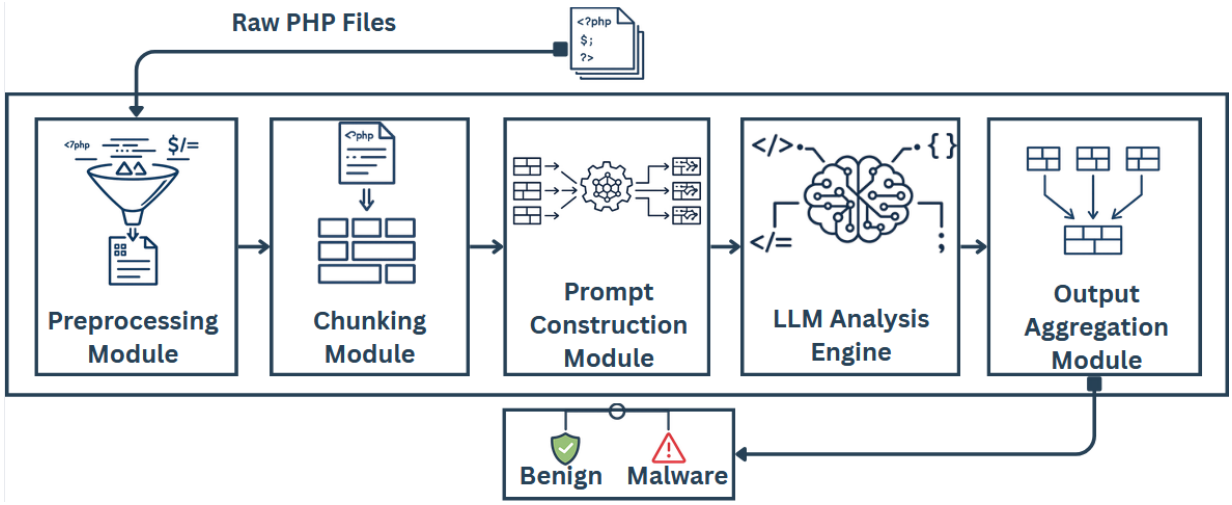

Fig. 1. Proposed Modular and Model-Agnostic PHP Webshell Detection Framework.

IV. Experimental Setup & Evaluation

A. Evaluation Metrics

To fairly evaluate the performance of the proposed detection framework, we use standard binary classification metrics, including Accuracy, Precision, Recall, and F1-score. These metrics measure how effectively the framework distinguishes between malicious and benign PHP files.

Since this is a binary classification task, the predictions can be summarized using a confusion matrix consisting of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). In the context of PHP webshell detection, these categories indicate how well the framework classifies malicious and benign files:

- **True Positives (TP)**: Malicious PHP files correctly classified as malicious.
- **True Negatives (TN)**: Benign PHP files correctly classified as benign.
- **False Positives (FP)**: Benign PHP files incorrectly classified as malicious.
- **False Negatives (FN)**: Malicious PHP files incorrectly classified as benign.

Using these values, the four evaluation metrics are defined as follows:

Accuracy measures the overall correctness of the framework:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Precision measures the proportion of files predicted as malicious that are actually malicious:

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

Recall measures the proportion of actual malicious files that are correctly identified:

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

F1-score provides a balanced measure of Precision and Recall:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

In PHP webshell detection, Recall is particularly important because a false negative means that a malicious backdoor remains undetected. This can allow the attacker to evade detection and maintain persistent compromise of the server and underlying operating system. However, an excessive number of false positives can also reduce practical usability by incorrectly flagging legitimate files as malicious. Therefore, a balanced evaluation across all metrics is necessary to assess the practical effectiveness of the framework.

B. Experimental Setup

The proposed malware detection framework was evaluated by integrating different large language models into the LLM analysis engine. Open-source LLMs such as Qwen2.5-Coder-7B-Instruct [17] and DeepSeek-Coder-7B-Instruct v1.5 [18] were selected because they are relatively lightweight coder models that can run on consumer-grade GPUs and are suitable for real-world deployment. Fine-tuning was also performed to examine whether model performance could be further improved. In addition to these open-source models, the framework was evaluated using Gemini 1.5 Flash [19] as a closed-source LLM for comparison.

Due to the lack of readily available datasets, we constructed a custom dataset consisting of both malicious and benign PHP files. Malicious PHP webshells were collected from three publicly available sources: the PHP Webshell Dataset [20], the Webshell Dataset by *c01dsnap* [21], and the PHP Webshell Dataset by *Cyc1e183* [22], which was compiled from multiple sources and referenced in *Pan et al.* [23]. After deduplication, the final malicious dataset contained 6,578 unique PHP webshells.

For benign PHP files, we used the *White.zip* set from the Webshell Dataset by *c01dsnap* [21]. From the 12,167 available files, 6,507 benign PHP files were selected through random sampling.

Both benign and malicious files were split into training and test sets using a 90:10 ratio. For the malicious files, the training set contained 5,921 samples and the test set contained 657 samples. For the benign files, the training set contained 5,850 samples and the test set contained 657 samples.

Only the test set, consisting of 657 malicious files and 657 benign files, was used for framework evaluation. Fine-tuning experiments were conducted using only the training set to examine whether recall could be further improved under the same framework.

C. Evaluation using Qwen2.5-Coder-7B-Instruct

The framework was first evaluated using Qwen2.5-Coder-7B-Instruct *(base)* as the LLM analysis engine. As shown in Table I, it delivered the most balanced overall performance among the evaluated open-source models, maintaining strong precision, recall, and F1-score while remaining lightweight and locally deployable. The prompt structure of the framework also enabled the model to generate short explanations for each prediction.

The fine-tuned version further improved recall, but this came at the cost of a substantial increase in false positives and a decline in the other evaluation metrics. This result highlights a clear trade-off between maximizing recall and maintaining balanced overall performance.

Overall, the results suggest that the base Qwen model is the more practical choice for general deployment, while the fine-tuned version may be useful in scenarios where recall is prioritized over precision.

TABLE I
PERFORMANCE COMPARISON OF EVALUATED METHODS

Method	Accuracy	Precision	Recall	F1-score
Regex	72.37	96.23	46.58	62.77
DeepSeek-Coder-7B v1.5	83.18	96.38	68.95	80.39
DeepSeek-Coder-7B v1.5 (FT)	92.39	88.63	97.26	92.74
Qwen2.5-Coder-7B	98.25	98.47	98.02	98.25
Qwen2.5-Coder-7B (FT)	89.95	83.61	99.39	90.82
Gemini 1.5 Flash	98.33	100.00	96.65	98.30

D. Evaluation using DeepSeek-Coder-7B-Instruct v1.5, Gemini 1.5 Flash, and Regex

The framework was then evaluated using DeepSeek-Coder-7B-Instruct v1.5 *(base)*. As shown in Table I, the base model showed relatively strong precision but weaker recall than Qwen. Fine-tuning substantially improved recall and overall performance, although it still did not outperform the Qwen base model.

A closed-source LLM, Gemini 1.5 Flash, was also evaluated within the same framework. It achieved the highest precision among all evaluated methods, indicating very strict classification behavior, although its recall remained slightly lower than that of the fine-tuned open-source models.

As a traditional baseline, regex-based detection was also evaluated. Compared with the LLM-based approaches, it showed particularly weak recall, suggesting that rule-based detection alone is insufficient for handling obfuscated or previously unseen webshells.

E. Overall Performance Comparison

A performance comparison among the evaluated open-source models, their fine-tuned versions, a closed-source model, and the traditional regex-based method is shown in Fig. 2. Overall, Qwen2.5-Coder-7B-Instruct *(base)* delivered the most balanced performance across the evaluated metrics while remaining locally deployable.

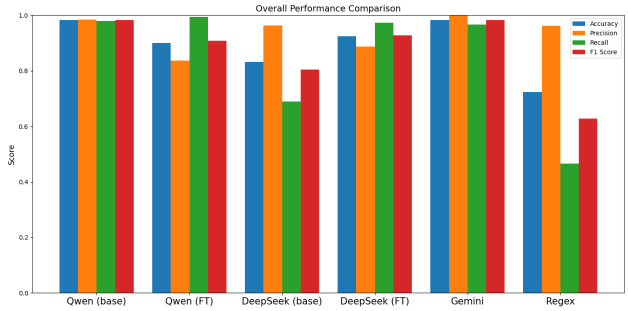

Fig. 2. Overall Performance Comparison

Across the evaluated methods, Qwen2.5-Coder-7B-Instruct *(fine-tuned)* achieved the highest recall, while Gemini 1.5 Flash achieved the highest precision. In contrast, the regex-based method showed the weakest overall performance, particularly in recall.

Taken together, the results highlight a clear trade-off between recall and precision. Fine-tuning improved sensitivity to malicious code but increased false positives, whereas the base Qwen model maintained the strongest balance across metrics. These comparisons suggest that context-aware LLM-based detection is more effective than traditional rule-based approaches for identifying obfuscated or previously unseen webshells.

F. Summary of Findings

These experimental results show that the proposed framework performs effectively across different large language models without requiring architectural changes. Open-source coder models can achieve performance close to proprietary services while still supporting full local deployment and preserving code confidentiality.

Among the evaluated methods, Qwen2.5-Coder-7B-Instruct *(base)* provided the most balanced performance among the open-source models. It achieved higher recall than Gemini 1.5 Flash while still maintaining strong precision and F1-score, all within a lightweight 7B-parameter model that can run locally. Fine-tuning can further improve recall and may be useful in scenarios where detecting all malicious files is prioritized over minimizing false positives. Larger models may further improve performance in environments with sufficient computational resources.

Closed-source LLMs such as Gemini may achieve higher precision, but they also raise privacy and compliance concerns when applied to sensitive codebases. Depending on vendor-specific data handling and retention policies, submitting source code to a proprietary API may introduce security and confidentiality risks. In contrast, locally deployed open-source models provide greater transparency and operational control.

Overall, these results suggest that the proposed framework provides a reliable, flexible, and model-agnostic approach to PHP webshell detection without dependence on proprietary services.

V. DISCUSSION

A. Strengths

The proposed framework offers several practical strengths for PHP malware detection. First, it can run locally or on a private server without requiring code to be sent to external services, which helps reduce privacy and compliance concerns.

The framework is also flexible and model-agnostic. Open-source large language models with strong code understanding capabilities can be used as the analysis engine without major changes to the rest of the pipeline. This allows the framework to support both smaller models in resource-constrained environments and larger models in settings with more available computational resources.

In addition, the framework provides context-aware analysis of PHP webshells and produces both a binary classification and a short explanation for each prediction. This supports integration with incident response workflows and related evaluation tools.

During evaluation, the framework was able to detect obfuscated and encoded malware, depending on the LLM used as the analysis engine.

Overall, the modular design, local deployability, and context-aware analysis make the framework adaptable to different detection scenarios and deployment environments.

B. Limitations and Future Work

Although the proposed framework shows strong performance and adaptability, it has several limitations. Processing large codebases can be relatively slow because long files must be divided into multiple chunks to fit within the model's context length. This can make detection more difficult when malicious behavior depends on long-range context or multi-stage execution logic distributed across multiple chunks. In resource-constrained environments, performance may also decline because smaller LLMs must be used, while larger models require greater computational resources.

The framework currently relies entirely on the LLM for classification, which can be computationally expensive. Lightweight pre-filtering methods, such as regex-based or signature-based checks, could help reduce the number of files requiring full LLM analysis. Similarly, file integrity monitoring could limit analysis to newly created or modified files, improving overall efficiency.

Future work may explore better chunk-merging strategies, incremental scanning, optional integration of opcode- or AST-based features, and model distillation or quantization to reduce resource requirements.

VI. Conclusion

This study proposed a modular and model-agnostic framework for PHP webshell detection built on open-source large language models. The framework enables context-aware analysis of raw PHP code while supporting local deployment in privacy-sensitive environments.

Experimental results showed that the framework can be adapted to different LLMs without architectural changes. Among the evaluated models, Qwen2.5-Coder-7B-Instruct achieved the most balanced overall performance while remaining lightweight and locally deployable. Fine-tuning improved recall but increased false positives, highlighting a trade-off between sensitivity and precision.

Overall, the results suggest that the proposed framework provides a practical, flexible, and privacy-preserving approach to PHP webshell detection. Its modular and model-agnostic design makes it suitable for deployment across different environments and operational requirements.

Acknowledgements

This research was partially supported by the National Science Foundation under Grant No. 2028011.

References

[1] E. A. Mann, *The PHP Cookbook*. Sebastopol, CA, USA: O'Reilly Media, 2023.

[2] Sucuri Inc., "2023 Hacked Website and Malware Threat Report," 2024. [Online]. Available: https://sucuri.net/reports/2023-hacked-website-report/

[3] Cybersecurity and Infrastructure Security Agency (CISA), "CISA adds four known exploited vulnerabilities to catalog," Jul. 7, 2025. [Online]. Available: https://www.cisa.gov/news-events/alerts/2025/07/07/cisa-adds-four-known-exploited-vulnerabilities-catalog

[4] N. Popov, "Changes to Git commit workflow," *PHP Mailing Lists*, Mar. 28, 2021. [Online]. Available: https://news-web.php.net/php.internals/113838

[5] Arya-f4, "worldshellfinder," GitHub. [Online]. Available: https://github.com/Arya-f4/worldshellfinder

[6] NSA Cybersecurity Directorate, "Mitigating-Web-Shells," GitHub. [Online]. Available: https://github.com/nsacyber/Mitigating-Web-Shells

[7] National Security Agency and Australian Signals Directorate, "Detect and Prevent Web Shell Malware," Apr. 21, 2020. [Online]. Available: https://media.defense.gov/2020/Jun/09/2002313081/-1/-1/0/CSI-DETECT-AND-PREVENT-WEB-SHELL-MALWARE-20200422.PDF

[8] Emposha, "PHP-Shell-Detector," GitHub. [Online]. Available: https://github.com/emposha/PHP-Shell-Detector/

[9] M. Ma, L. Han, and C. Zhou, "Research and application of artificial-intelligence-based webshell detection model: A literature review," arXiv:2405.00066, 2024.

[10] N. H. Nguyen, V. H. Le, V. O. Phung, and P. H. Du, "Toward a deep learning approach for detecting PHP webshell," in *Proc. 10th Int. Symp. Inf. Commun. Technol.*, Dec. 2019, pp. 514–521.

[11] Y. Tian, J. Wang, Z. Zhou, and S. Zhou, "CNN-webshell: Malicious web shell detection with convolutional neural network," in *Proc. 2017 VI Int. Conf. Netw., Commun. Comput.*, Dec. 2017, pp. 75–79.

[12] Z. Zhou, L. Li, and X. Zhao, "Webshell detection technology based on deep learning," in *Proc. 2021 7th IEEE Int. Conf. Big Data Security on Cloud (BigDataSecurity), HPSC, and IDS*, May 2021, pp. 52–56, doi: 10.1109/BigDataSecurityHPSCIDS52275.2021.00020.

[13] B. Gogoi, T. Ahmed, and R. G. Dinda, "PHP web shell detection through static analysis of AST using LSTM-based deep learning," in *Proc. 1st Int. Conf. Artif. Intell. Trends Pattern Recognit. (ICAITPR)*, Mar. 2022, pp. 1–6.

[14] Y. Fang, S. Li, and Z. Hu, "Webshell detection by transformer model," in *Proc. 2025 5th Int. Symp. Comput. Technol. Inf. Sci. (ISCTIS)*, May 2025, pp. 482–485.

[15] Y. Li, J. Huang, A. Ikusan, M. Mitchell, J. Zhang, and R. Dai, "ShellBreaker: Automatically detecting PHP-based malicious web shells," *Comput. Secur.*, vol. 87, Art. no. 101595, 2019, doi: 10.1016/j.cose.2019.101595.

[16] L. Bai and Y. Zhu, "WebshellHunter: A new webshell detection method based on abstract syntax tree and CNN–BiLSTM," in *Proc. 5th Int. Conf. Comput. Netw. Security Softw. Eng.*, Feb. 2025, pp. 356–362.

[17] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, and J. Lin, "Qwen2.5-Coder technical report," arXiv:2409.12186, 2024.

[18] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, and W. Liang, "DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence," arXiv:2401.14196, 2024.

[19] Gemini Team, P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati, and B. O. Batsaikhan, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," arXiv:2403.05530, 2024.

[20] Z. Wang, H. Wang, and L. Hao, "Poster: Long PHP webshell files detection based on sliding window attention," arXiv:2502.19257, 2025.

[21] c01dsnap, "Webshell dataset," Hugging Face, 2024. [Online]. Available: https://huggingface.co/datasets/c01dsnap/Webshell

[22] Cyc1e183, "PHP webshell dataset," GitHub, 2021. [Online]. Available: https://github.com/Cyc1e183/PHP-Webshell-Dataset

[23] Z. Pan, Y. Chen, Y. Chen, Y. Shen, and X. Guo, "Webshell detection based on executable data characteristics of PHP code," *Wireless Commun. Mobile Comput.*, vol. 2021, Art. no. 5533963, 2021.