Can LLMs handle WebShell detection? Overcoming Detection Challenges with Behavioral Function-Aware Framework

Feijiang Han[*]
Jiaming Zhang[†] Chuyi Deng[‡]
Jianheng Tang[§] Yunhuai Liu[¶]

Abstract

WebShell attacks, in which malicious scripts are injected into web servers, are a major cybersecurity threat. Traditional machine learning and deep learning methods are hampered by issues such as the need for extensive training data, catastrophic forgetting, and poor generalization. Recently, Large Language Models (LLMs) have gained attention for code-related tasks, but their potential in WebShell detection remains underexplored. In this paper, we make two major contributions: (1) a comprehensive evaluation of seven LLMs, including GPT-4, LLaMA 3.1 70B, and Qwen 2.5 variants, benchmarked against traditional sequence- and graph-based methods using a dataset of 26.59K PHP scripts, and (2) the Behavioral Function-Aware Detection (BFAD) framework, designed to address the specific challenges of applying LLMs to this domain. Our framework integrates three components: a Critical Function Filter that isolates malicious PHP function calls, a Context-Aware Code Extraction strategy that captures the most behaviorally indicative code segments, and Weighted Behavioral Function Profiling (WBFP) that enhances in-context learning by prioritizing the most relevant demonstrations based on discriminative function-level profiles. Our results show that larger LLMs achieve near-perfect precision but lower recall, while smaller models exhibit the opposite trade-off. However, all models lag behind previous State-Of-The-Art (SOTA) methods. With BFAD, the performance of all LLMs improved, with an average F1 score increase of 13.82%. Larger models such as GPT-4, LLaMA 3.1 70B, and Qwen 2.5 14B outperform SOTA methods, while smaller models such as Qwen 2.5 3B achieve performance competitive with traditional methods. This work is the first to explore the feasibility and limitations of LLMs for WebShell detection, and provides solutions to address the challenges in this task.

1 Introduction

The rapid growth of web applications and the expansion of cloud-based services have escalated the cybersecurity threat landscape, with WebShells emerging as a significant concern. WebShells are malicious scripts injected into web servers that allow attackers to remotely execute arbitrary commands, steal sensitive data, and compromise system integrity (Starov et al., 2016; Cui et al., 2018). According to a Cisco Talos report (Talos, 2024), threat groups used a variety of web shells against vulnerable or unpatched web applications in 35% of incidents in Q4 2024, a sharp increase from the previous quarter, when such activity was observed in only 10% of incidents. These attacks are particularly dangerous due to the stealthy nature of WebShells, which are rapidly evolving to evade traditional detection methods (Hannousse & Yahiouche, 2021).

In response to this challenge, the community has proposed several approaches. Rule-based methods that rely on predefined signatures or heuristics are increasingly ineffective against the complexity and diversity of modern WebShells (Le et al., 2021; Jinping et al., 2020). Machine learning models,

[*]University of Pennsylvania, feijhan@seas.upenn.edu
[†]Central South University, 8213200215@csu.edu.cn
[‡]Central South University, 8208220112@csu.edu.cn
[§]Peking University, tangentheng@stu.pku.edu.cn, corresponding author
[¶]Peking University, yunhuai.liupku.edu.cn, corresponding author

especially deep learning techniques (Pu et al., 2022), have shown promise in addressing these threats. However, they require extensive training on large datasets - resources that are often difficult to obtain and sensitive in nature (Shang et al., 2024). In addition, these models face challenges such as catastrophic forgetting and poor generalization, especially when dealing with obfuscated or encrypted attacks (Jinping et al., 2020; Zhang et al., 2025).

Recently, Large Language Models (LLMs) have gained attention for a variety of code-related tasks, including code generation (Ma et al., 2024) and vulnerability detection (Liu & He, 2023; Wang et al., 2025). Studies have shown that with timely engineering, LLMs can perform remarkably well without additional training (Nong et al., 2024; Trad & Chehab, 2025). They also provide interpretable explanations for their decisions, which is critical in cybersecurity contexts (Ma et al., 2024). Despite these advantages, there has been limited exploration of LLMs for WebShell detection.

Detecting WebShells with LLMs presents unique challenges that differ from other code analysis tasks. WebShells often employ obfuscation and encryption techniques, and are embedded in large codebases dominated by benign content (Liu & He, 2023). For example, the largest WebShell in our dataset spans 1,386,438 tokens, far exceeding the context window of most LLMs, which risks truncating critical malicious segments when processing entire source files (Wang et al., 2025; Ceka et al., 2024). In addition, in-context learning (ICL) struggles in this domain: the variability and obfuscation of WebShells complicates the selection of effective demonstrations, and these examples further occupy significant context space, reducing capacity for the target code (Yuan et al., 2024). While recent research has focused on increasing the context length of LLMs (Chen et al., 2023), studies suggest that the performance of an LLM tends to degrade with longer inputs, and the low processing speed may also become unacceptable for practical use (Ma et al., 2024; Fang et al., 2024).

In this paper, we present two key contributions to advance the application of LLMs in WebShell detection.

First, we systematically evaluate LLMs in the context of WebShell detection, comparing their performance with traditional state-of-the-art machine learning (SOTA) methods. Specifically, we evaluate seven closed-source and open-source LLMs of different sizes, including GPT-4 (Achiam et al., 2023), LLaMA 3.1 70B (Grattafiori et al., 2024), Qwen 2.5 Coder (14B/3B) (Yang et al., 2024), and Qwen 2.5 (3B/1.5B/0.5B) (Yang et al., 2024) on a dataset containing 26.59K PHP scripts (4.93K WebShells and 21.66K benign samples). Our analysis reveals several key findings:

- Larger LLMs, such as GPT-4 and Qwen 2.5 Coder 14B, achieve near perfect precision (close to 100%), but struggle with recall (e.g., GPT-4's recall is 85.98%), indicating difficulty in detecting all malicious instances.
- Smaller LLMs, such as Qwen 2.5 Coder 3B and Qwen 2.5 0.5B, have high recall (close to 100%) but suffer from low precision (e.g. Qwen 2.5 Coder 3B's precision is 38.93%), indicating a tendency to misclassify benign files as malicious.
- Randomly selected ICL demonstrations degrade LLM detection performance; examples selected based on semantic similarity to the source code do not yield significant improvements.

We compare the performance of LLMs with traditional methods, including Glove+SVM (Petridis, 2024; Rigutini et al., 2024), CodeBERT+Random Forest (Alghamdi et al., 2022), and graph-based approaches such as GAT (Kang et al., 2023). The best performing LLM, Qwen 2.5 Coder 14B, achieves an F1 score of 96.39%, although it still lags behind GAT-based methods, which achieve an F1 score of 98.87%.[1].

Second, we present the Behavioral Function-Aware Detection (BFAD) framework to address the identified challenges and to improve the performance of LLMs to meet the requirements of downstream applications by achieving a more balanced trade-off between precision and recall. Our framework combines risk-filtering techniques with an enhanced ICL strategy that uses weighted demonstration selection to prioritize examples most closely related to key malicious behaviors. Experimental results show that our approach improves the average F1 score across all LLMs by 13.82%, with GPT-4

[1]While traditional models such as GAT have demonstrated superior performance, they require extensive training and significant data collection efforts. In contrast, LLMs enable direct detection through prompts, leveraging their pre-trained knowledge with minimal task-specific resources to deliver competitive results.

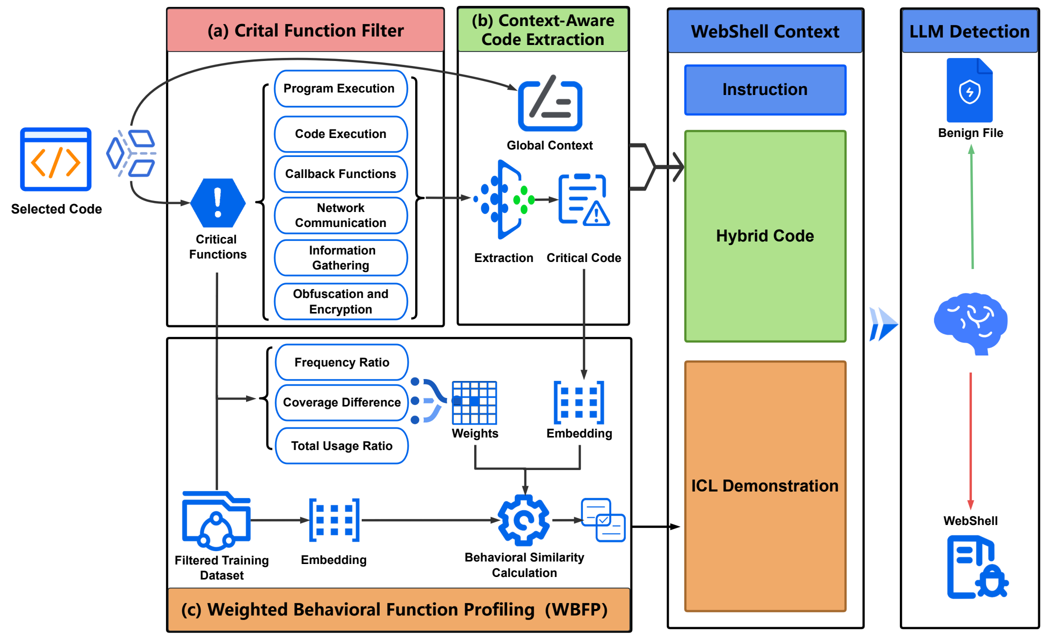

Figure 1: Overview of the Behavioral Function-Aware Detection framework for WebShell detection. It consists of three components: (a) Critical Function Filter, which identifies PHP functions associated with malicious behavior; (b) Context-Aware Code Extraction, which isolates critical code regions to overcome LLM context limitations; and (c) Weighted Behavioral Function Profiling, which selects ICL demonstrations using a behavior-weighted similarity score.

and Qwen 2.5 0.5B improving by 6.89% and 51.32%, respectively. In several cases, such as GPT-4, LLaMA 3.1 70B, Qwen 2.5 Coder 14B, and Qwen 2.5 Coder 3B, our approach enables performance that is competitive or even superior to traditional methods.

To the best of our knowledge, this is the first work to systematically analyze the feasibility and limitations of applying LLMs to WebShell detection.

2 Behavioral Function-Aware Detection Framework

We present the **Behavioral Function-Aware Detection (BFAD)** framework, a comprehensive solution designed to improve WebShell detection by identifying critical code segments and improving the quality of ICL demonstration selection. The architecture of BFAD, as shown in Figure 1, consists of three primary components: (a) **Critical Function Filter**, which is based on malicious behavior patterns and filters critical PHP function calls to identify key functions; (b) **Context-Aware Code Extraction**, which addresses the limitations of LLMs in handling long input sequences by selectively extracting critical code regions; and (c) **Weighted Behavioral Function Profiling**, which improves demonstration selection by calculating a weighted similarity score based on function-level profiling.

2.1 Critical Function Filter

WebShells typically rely on specific PHP functions that facilitate malicious actions such as code execution, data exfiltration, or obfuscation. However, these functions are often embedded in complex, obfuscated code, making detection difficult. To address this, we are developing a **Critical Function Filter** that classifies PHP functions into six different behavioral categories: *Program Execution*, *Code Execution*, *Callback Functions*, *Network Communication*, *Information Gathering*, and *Obfuscation and Encryption*. This taxonomy reflects the different roles these functions play in WebShells.

Specifically, the **Program Execution** category includes functions such as `exec` and `system`, which execute system-level commands and are often exploited in WebShells to execute arbitrary

Algorithm 1 Context-Aware Code Extraction

Input: Source code $\mathcal{C}$, list of critical functions $\mathcal{F}$, context window size τ
Output: Extracted critical code regions $\mathcal{R}$
Initialize empty set of regions: $\mathcal{R} \leftarrow \emptyset$
for each function $f \in \mathcal{F}$ **do**
    Locate all occurrences of f in $\mathcal{C}$
    for each occurrence of f at position p **do**
        Extract context window $[p - \tau, p + \tau]$ from $\mathcal{C}$
        Add the extracted region to $\mathcal{R}$
    end for
end for
Merge overlapping regions in $\mathcal{R}$
Compute remaining context budget B
if $B > 0$ **then**
    Select additional non-overlapping code segments from $\mathcal{C}$
    Add selected segments to $\mathcal{R}$
end if
return $\mathcal{R}$

payloads. Similarly, **Code Execution** functions such as `eval` and `preg_replace` interpret input as executable code, allowing attackers to inject arbitrary scripts. The **Callback Functions** category includes functions such as `array_map` and `register_shutdown_function`, which allow dynamic invocation of functions often used to obfuscate malicious code.

In addition, **Network Communication** functions such as `fsockopen` and `curl_init` allow remote communication for data exfiltration and command-and-control operations. **Information Gathering** functions, such as `phpinfo` and `getenv`, are used by attackers to gather system details. Finally, **Obfuscation and Encryption** functions, such as `base64_encode` and `openssl_encrypt`, help disguise or encrypt malicious payloads to avoid detection.

Our statistical analysis (detailed in Appendix B) shows that WebShell files use critical functions far more often than benign files. On average, WebShells contain 22.76 calls to critical functions, compared to only 0.74 in benign files, underscoring their behavioral complexity and distinctiveness.[2]

2.2 Context-Aware Code Extraction

Building on the Critical Function Filter, we introduce a **Context-Aware Code Extraction** strategy that identifies and extracts the critical code regions that indicate malicious behavior. These regions focus on the identified critical functions and their surrounding contexts, ensuring that the LLM focuses on the most relevant parts of the code.

The complete extraction procedure is formalized in Algorithm 1, which takes as input the source code $\mathcal{C}$, the list of critical functions $\mathcal{F}$, and the context window size τ, and produces a set of extracted critical code regions $\mathcal{R}$.

We reduce the input size by selectively extracting critical regions of code and merging overlapping segments while preserving behavioral specificity. However, this approach may inadvertently exclude certain global contextual information and increase the emphasis on critical functions, potentially leading to false positives when analyzing benign files that legitimately use such functions. To mitigate this, we append truncated, non-overlapping code segments when context length allows, ensuring that the model receives a balanced representation of local and global code context.

2.3 Weighted Behavioral Function Profiling

Based on the extracted critical code regions, we propose **Weighted Behavioral Function Profiling (WBFP)**, a method that computes a weighted similarity score to identify behaviorally similar examples

[2] While this disparity in critical function usage is significant, it alone does not reliably distinguish WebShells from benign files, since legitimate scripts may also call these functions. Therefore, we use LLMs for deeper contextual analysis to improve detection accuracy.

for ICL effectively. WBFP assigns weights to each function type based on its prevalence and usage in WebShell versus benign files, quantified by three metrics: coverage difference (r_c), frequency ratio (r_f), and usage ratio (r_u). The coverage difference measures the proportion of files containing a specific function across the two datasets. The frequency ratio is the ratio of the average number of occurrences of the function per file in WebShell files to the average number per file in benign files. The usage ratio reflects the total number of function occurrences in WebShell files compared to those in benign files. These metrics are combined to calculate a discrimination score for each function type

$$\text{Score}_f = (r_c \cdot \alpha) + (r_f \cdot \beta) + (r_u \cdot \gamma),$$

where α, β, and γ are empirically determined weights, set to 1 for balanced contribution in our experiments (see Section 3.2). We normalize the discrimination scores to weights:

$$w_f = \frac{\text{Score}_f}{\sum_{f' \in \mathcal{F}} \text{Score}_{f'}}.$$

WBFP then uses the embeddings $E(\cdot)$ generated by *st-codesearch-distilroberta-base* (Abi Akl, 2023; Al-Kaswan et al., 2023) to compute the similarity between the files x and y. Let $\mathcal{F}$ denote all critical function types. For each function type $f \in \mathcal{F}$, we concatenate critical regions $R_f(x)$ and compute their embeddings:

$$\mathbf{e}_f(x) = E(\text{concat}_f(x)),$$

The semantic similarity between the files x and y for the function type f is given by

$$s_f(x, y) = \frac{\mathbf{e}_f(x) \cdot \mathbf{e}_f(y)}{\|\mathbf{e}_f(x)\| \|\mathbf{e}_f(y)\|}.$$

The final similarity between the files is the weighted sum of the similarities:

$$\text{Sim}(x, y) = \sum_{f \in \mathcal{F}} w_f \cdot s_f(x, y).$$

This weighted similarity prioritizes function types critical to WebShell behavior, reducing the impact of irrelevant semantic features. Using this score, WBFP ensures that ICL demonstrations accurately capture malicious patterns, improving detection performance.

2.4 LLM-Based Detection Framework

We integrate the BFAD framework into the LLM-based detection system, which combines the context-aware code extraction strategy and WBFP to optimize the use of LLMs for WebShell detection. By incorporating both critical code regions and global context, along with behaviorally relevant demonstrations, our framework enhances the ability of the LLM to accurately identify malicious code patterns.

The input to the LLM consists of two main components: (a) a **system directive** that defines the model's role as a cybersecurity expert, and (b) a **user query** that contains the extracted critical code segments and a selected ICL demonstration. To balance efficiency and performance, we limit the user query to one ICL demonstration, which reduces computational overhead while preserving sufficient context for reliable detection. Our prompt is described in detail in Appendix A.

3 Experiment

3.1 Dataset Overview

We constructed a comprehensive dataset consisting of 21,665 benign PHP programs and 4,929 webshells. The benign programs were obtained from established open-source PHP projects to ensure applicability to real-world scenarios. The webshells were collected from public security repositories and augmented with synthetic obfuscation techniques to increase diversity. [3] Using GPT-4's tokenizer, we analyzed the token lengths of both sample types. The webshell samples had a maximum token length of 1,386,438 and an average of 30,856.60 tokens, compared to a maximum of 305,670 tokens and an average of 2,242.89 tokens for benign programs. These results indicate that webshells are typically significantly longer than benign samples. A detailed summary of the dataset composition can be found in the Table 2 in the Appendix C.

3.2 Experiment Setup

ICL Settings. We randomly selected 60% of the dataset to create a fixed known demonstration library for ICL. Using this subset, we computed normalized scores for different function categories based on the WBFP method, giving equal weight to coverage difference (r_c), frequency ratio (r_f), and usage ratio (r_u) to profile functions according to their behavioral importance in distinguishing webshells from benign programs. These scores, detailed in the Table 3 in the Appendix C, guided the selection of ICL demonstrations from the known sample library.

Baseline Models, Hyperparameter Settings, and Evaluation Metrics. We compared our approach to several baselines: GloVe + SVM, CodeBERT + Random Forest, GCN, and GAT. For GloVe + SVM, we used pre-trained GloVe embeddings with a dimensionality of 300 and an SVM classifier with default parameters (Qi et al., 2018; ZENG et al., 2025). For CodeBERT + Random Forest, we used CodeBERT embeddings with a hidden dimension of 768 and a Random Forest classifier with default settings (Wang et al., 2024a). The GCN was trained with a learning rate of 0.001 over 120 epochs, with 3 hidden layers and a hidden dimension of 32 (Feng et al., 2024). The GAT was trained with a learning rate of 0.001 over 120 epochs, with 3 hidden layers, a hidden dimension of 8, and 8 attention heads (Feng et al., 2024). We evaluated the models using standard classification metrics: accuracy, precision, recall, and F1 score.

4 Results and Analysis

In this section, we systematically evaluate the performance of LLMs of different scales for WebShell detection, and assess the improvements provided by our proposed BFAD framework. Our analysis addresses three research questions (RQs) to explore both the baseline LLM capabilities and the effectiveness of the BFAD components:

- **RQ1:** How do large and small scale LLMs perform in WebShell detection compared to traditional ML and DL methods, and how does BFAD improve their effectiveness?
- **RQ2:** How effective is context-aware code extraction at balancing global context and local behavioral focus under LLM context length constraints?
- **RQ3:** How does WBFP improve demo selection for ICL?

4.1 Performance Evaluation of LLMs and BFAD Enhancements (RQ1)

We evaluated seven LLMs, including large-scale models such as GPT-4, LLaMA-3.1-70B, and Qwen-2.5-Coder-14B, and small-scale models such as Qwen-2.5-Coder-3B, Qwen-2.5-3B, Qwen-2.5-1.5B, and Qwen-2.5-0.5B. These were compared with traditional ML and DL baselines, including sequence-based methods (Glove+SVM, CodeBERT+RF) and graph-based methods (GCN, GAT). The results are detailed in Table 4 in Appendix D.

Characteristics of LLMs in WebShell Detection Our evaluation reveals distinct performance characteristics of LLMs in WebShell detection that are influenced by their size. Compared to traditional machine learning and deep learning methods, especially GAT, vanilla LLMs exhibit

[3] We acknowledge the potential for data leakage due to the pre-training of LLMs. However, the autoregressive nature of this process does not explicitly capture WebShell classification, although it may affect other code generation tasks. To further minimize any risk of leakage, we restricted our selection of PHP programs to projects updated between October 2024 and 2025, thus reducing overlap with LLM training corpora.

unbalanced performance - large models lack sufficient recall, while small models lack precision - making them less effective without targeted improvements.

In particular, large LLMs such as GPT-4 and Qwen-2.5-Coder-14B achieve high precision (100% and 99.32%, respectively) but moderate recall (85.98% and 93.63%, respectively), reflecting a conservative classification bias. This is likely due to their extensive training on diverse datasets, which prioritizes accurate identification of benign code over detection of rare or novel WebShells. Code-specific fine-tuning improves the performance of Qwen-2.5-Coder-14B over general-purpose models such as LLaMA-3.1-70B (97.31% precision, 92.36% recall), underscoring the benefits of domain specialization. In contrast, small-scale LLMs such as Qwen-2.5-0.5B and Qwen-2.5-1.5B exhibit nearly perfect recall (100% and 95.77%, respectively) but extremely low precision (18.65% and 34.61%, respectively). This imbalance may be due to their increased sensitivity to prompts, making them more likely to provide answers that match user expectations, a problem exacerbated by their limited ability to model complex code relationships.

Effects of BFAD on LLMs The BFAD framework significantly improves the performance of LLMs for WebShell detection. For large-scale LLMs, BFAD increases recall while maintaining near-perfect precision; in particular, GPT-4's recall increases by 12.73% to 98.71%, yielding an F1 score of 99.35%, which exceeds the graph-based GAT baseline (98.87% F1), while LLaMA-3.1-70B achieves an F1 score of 98.40%, closely matching GAT. Conversely, for small LLMs, BFAD significantly increases precision without compromising high recall; Qwen-2.5-0.5B's precision increases by 52.36% to 71.10%, resulting in an F1 score of 82.67% (a 51.23% improvement), and Qwen-2.5-1.5B achieves an F1 score of 65.33% (up 14.85%). These advances enable BFAD-enhanced LLMs to match or exceed traditional methods without the need for additional fine-tuning or training, driven by a strategic focus on critical code regions and the use of weighted behavioral profiling to select contextually relevant demonstrations. This approach effectively mitigates the inherent context length constraints of LLMs and improves their generalization from limited examples. Notably, small-scale LLMs exhibit more pronounced gains, underscoring the critical role of structured input for models with limited capacity, while large-scale LLMs benefit from improved recall, making them suitable for safety-critical applications. These results highlight the transformative potential of integrating domain-specific strategies with LLMs to address specialized challenges, such as WebShell detection, with exceptional effectiveness.

4.2 The Effectiveness of Context-Aware Code Extraction (RQ2)

We evaluated the effectiveness of Context-Aware Code Extraction using two models: GPT-4, a large-scale model, and Qwen-2.5-3B, a smaller-scale model. Three configurations were compared: (1) predictions based on the full source code, (2) predictions using only extracted critical regions, and (3) a hybrid approach combining critical regions with truncated source code. Results are reported in Tables 5 and 6 in Appendix D.

Impact of Critical Regions For the smaller model, Qwen-2.5-3B, critical regions significantly improve performance over the full source baseline. At $\tau = 100$, the F1 score increases from 84.37% to 90.91% (+6.54%), with precision increasing from 78.03% to 86.71% (+8.68%) and recall increasing from 91.84% to 95.54% (+3.70%). This improvement validates the focus on behaviorally relevant code snippets, which reduces irrelevant context and sharpen the focus of the model. However, as the context length (τ) increases, performance decreases - F1 drops to 88.17% at $\tau = 300$, likely due to the model's limited ability to handle extended context, which introduces noise that degrades accuracy.

For the larger model, GPT-4, critical regions increase recall but slightly decrease precision. At $\tau = 300$, recall improves from 85.98% to 96.18% (+10.20%), while precision drops from 100.00% to 98.69% (-1.31%). The F1 score increases from 92.46% to 97.42% (+4.96%), indicating that GPT-4 effectively uses localized behavioral cues to improve recall. However, the reduced global context may introduce small biases, leading to a precision trade-off, although the overall performance remains strong due to the model's greater capacity.

Balancing Precision and Recall with the Hybrid Strategy The hybrid strategy improves model performance by effectively balancing precision and recall. For Qwen-2.5-3B, it increases precision over using critical regions alone, from 86.71% to 89.02% (+2.31%) for $\tau = 100$ and from 82.32% to 85.55% (+3.23%) for $\tau = 300$. Although recall decreases slightly, the F1 score increases to 89.70% at $\tau = 300$ (+1.53% from 88.17%), indicating that the hybrid approach reduces noise in longer contexts and supports smaller models by maintaining focus on critical regions while integrating valuable global context. For GPT-4, the strategy improves recall without compromising precision: at $\tau = 300$, recall increases from 85.98% (using the source code) to 96.82% (+10.84%), while precision remains steady at 100.00%, yielding an F1 score of 98.38%–a 5.92% gain over the source code and a 0.96% improvement over critical regions alone.

4.3 The Effectiveness of WBFP for In-Context Learning (RQ3)

We evaluated the effectiveness of WBFP for ICL demonstration selection using Qwen-2.5-3B and GPT-4. This evaluation builds on the Context-Aware Code Extraction with the hybrid strategy. Five demonstration selection strategies were compared: Random Selection (Random), Source Code Semantic Similarity (SC-Sim), WBFP with Equal Weights (WBFP-Eq), and WBFP with Function-Level Weights (WBFP-Wt). Results are detailed in Tables 7 and 8 in Appendix D.

Limitations of Random and Semantic Similarity-Based Selection Random selection makes ICL performance much worse by adding irrelevant examples, which makes both models much less effective. For Qwen-2.5-3B, the F1 score drops to 60.83% under the hybrid strategy with a context length of $\tau = 100$, which is a 23.54% reduction from the no-ICL baseline of 84.37%. This is mainly because there is a large drop in precision from 78.03% to 46.33%, although recall remains strong at 88.53%. Similarly, for GPT-4 with $\tau = 300$, the F1 score drops to 76.22%, a 22.16% decrease from the No-ICL baseline of 98.38%, driven by a drop in precision from 100.00% to 65.58%. These results indicate that Random Selection does not provide contextually relevant demonstrations, making it ineffective for WebShell detection.

The SC-Sim approach, which relies on semantic similarity calculated over entire source code samples, also doesn't work well. For Qwen-2.5-3B, SC-Sim achieves an F1 score of 84.36%, which is almost the same as the No-ICL baseline (84.37%), with a precision of 78.57% and a recall of 91.08%. For GPT-4, it achieves an F1 score of 96.32%, with perfect precision (100.00%), but a reduced recall of 92.90% compared to the No-ICL baseline's 96.82%. This limited performance is likely due to the dominance of behaviorally irrelevant code segments in the similarity calculation, which dilutes the focus on critical behavioral patterns essential for accurate WebShell identification.

Superiority of WBFP in Demonstration Selection For both models, WBFP-Wt consistently outperforms in accuracy, precision, recall, and F1 score, demonstrating its robustness and adaptability for improving ICL in WebShell detection tasks. Specifically, for Qwen-2.5-3B, WBFP-Wt achieves an F1 score of 93.69%, which is 9.33% higher than SC-Sim and 1.28% higher than WBFP-Eq. This is because WBFP-Wt achieves a precision of 88.64% (compared to 78.57% for SC-Sim and 86.39% for WBFP-Eq) and a near perfect recall of 99.36%. By focusing on the important parts of WebShell behavior, WBFP-Wt makes up for the fact that the smaller model doesn't understand as much. This results in demonstrations that closely match the desired behavioral profiles, improving both precision and recall.

For GPT-4, WBFP-Wt achieves the highest F1 score of 99.35%, which is 3.03% higher than SC-Sim and 0.34% higher than WBFP-Eq . While precision remains at 100.00% across all WBFP variants, WBFP-Wt increases recall to 98.71%, compared to 92.90% for SC-Sim and 98.03% for WBFP-Eq. This improvement highlights WBFP-Wt's ability to leverage GPT-4's advanced understanding of context, matching selected demonstrations to the behavioral characteristics of the target sample to optimize recall without sacrificing precision.

5 Related Work

WebShell Detection Techniques Early efforts in WebShell detection relied on rule-based methods that used signature matching or heuristics to identify malicious scripts (Le et al., 2021; Jinping et al., 2020). For example, Le et al. (2021) proposed H-DLPMWD, a hybrid approach that combines pattern matching with a CNN to detect ASP.NET WebShells, achieving 98.49% accuracy by using Yara-based filtering and opcode indexing. However, such methods struggle against obfuscated or novel WebShell variants due to their reliance on predefined patterns (Hannousse & Yahiouche, 2021). Machine learning (ML) techniques have advanced this landscape by extracting features from code text or runtime behavior. Jinping et al. (2020) introduced a mixed-model approach using Random Forest and CNNs with N-gram and TF-IDF features that achieved 97% accuracy on PHP WebShells, but it requires balanced datasets and struggles with encrypted samples. Deep learning (DL) has further improved adaptability, with models such as CodeBERT for semantic analysis (Pu et al., 2022) and Graph Attention Networks (GAT) for structural insights (Zhang et al., 2025). (Zhang et al., 2025) proposed MMFDetect, which fuses CodeBERT-CL semantics with CNN-extracted visual features from RGB-mapped PHP code, achieving 99.47% accuracy on evasive WebShells. Despite these gains, ML and DL methods require extensive labeled data-a scarcity in cybersecurity-and exhibit limited generalization to obfuscated threats, along with high computational cost (Shang et al., 2024; Jinping et al., 2020).

LLMs in Code Analysis LLMs have revolutionized code-related tasks by leveraging large pre-training corpora for applications such as code generation (Ma et al., 2024), vulnerability detection (Sun et al., 2024), and program reliability assessment (Liu et al., 2024). Ma et al. (2024) demonstrated the ability of LLMs to generate evasive WebShells using hybrid prompts, highlighting their code synthesis potential. Sun et al. (2024) introduced LLM4Vuln, which enhances vulnerability reasoning through knowledge retrieval and has achieved practical success in Solidity audits. These models excel at zero-shot and few-shot learning via prompt engineering (Nong et al., 2024), providing interpretability critical for security contexts (Ma et al., 2024). However, their application to WebShell detection remains underexplored, with previous studies focusing on generation or general vulnerabilities rather than detection of stealthy, context-heavy WebShells.

Challenges of LLMs for WebShell Detection Using LLMs for WebShell detection reveals critical bottlenecks. The fixed context window truncates large WebShells, potentially missing malicious segments embedded in benign code (Ceka et al., 2024; Wang et al., 2025). Fang et al. (2024) found that LLM performance degrades with longer inputs, with GPT-4's accuracy dropping to 87% on obfuscated JavaScript. Solutions such as chunking or sparse attention (e.g., LongCoder (Guo et al., 2023), SparseCoder (Wang et al., 2024b)) mitigate this, but often lose global context (Wang et al., 2025). In-context learning (ICL), a cornerstone of LLM adaptability, falters because demonstrations consume context space (Min et al., 2022; Wang et al., 2025), and random or semantic similarity-based selections fail to capture WebShell-specific behaviors. Liu & Wang (2023) proposed maximum information gain for ICL, but its focus on text classification limits its applicability to code security.

These gaps between rule-based rigidity, ML and DL data dependency, and LLM context and ICL limitations motivate our BFAD framework. BFAD overcomes context constraints with a hybrid extraction strategy that preserves critical regions and global cues, outperforming naive LLM applications. In addition, our WBFP enhances ICL by prioritizing behaviorally relevant demonstrations, outperforming generic similarity-based selection methods.

Conclusion

This paper presents a novel approach to WebShell detection using LLMs, addressing the unique challenges of applying these models to cybersecurity. We evaluated seven LLMs, including GPT-4 and LLaMA 3.1 70B, against traditional machine learning and deep learning methods using a dataset of 26.59K PHP scripts. Our analysis revealed that larger LLMs achieve near-perfect precision but lower recall, while smaller models exhibit high recall but poor precision, both of which underperform state-of-the-art methods such as GAT. To overcome these limitations, we proposed the BFAD framework, which integrates a critical function filter, context-aware code extraction, and WBFP. BFAD significantly improved LLM performance, with GPT-4 achieving an F1 of 99.35%, surpassing traditional benchmarks, and smaller models such as Qwen-2.5-0.5B improving by 51.23 percentage points to an F1 of 82.67%. As the first systematic exploration of LLMs for WebShell detection, this work not only demonstrates their potential, but also provides actionable solutions to their contextual and behavioral challenges, paving the way for future advances in code security.

References

Hanna Abi Akl. A ml-llm pairing for better code comment classification. In *FIRE (Forum for Information Retrieval Evaluation) 2023*, 2023.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. Stacc: Code comment classification using sentencetransformers. In *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pp. 28–31. IEEE, 2023.

Jawaher Alghamdi, Yuqing Lin, and Suhuai Luo. A comparative study of machine learning and deep learning techniques for fake news detection. *Information*, 13(12):576, 2022.

Ira Ceka, Feitong Qiao, Anik Dey, Aastha Valecha, Gail Kaiser, and Baishakhi Ray. Can llm prompting serve as a proxy for static analysis in vulnerability detection. *arXiv preprint arXiv:2412.12039*, 2024.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

Handong Cui, Delu Huang, Yong Fang, Liang Liu, and Cheng Huang. Webshell detection based on random forest–gradient boosting decision tree algorithm. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pp. 153–160. IEEE, 2018.

Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, Houman Homayoun, et al. Large language models for code analysis: Do {LLMs} really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 829–846, 2024.

Pengbin Feng, Dawei Wei, Qiaoyang Li, Qin Wang, Youbing Hu, Ning Xi, and Jianfeng Ma. Glareshell: Graph learning-based php webshell detection for web server of industrial internet. *Computer Networks*, 245:110406, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*, pp. 12098–12107. PMLR, 2023.

Abdelhakim Hannousse and Salima Yahiouche. Handling webshell attacks: A systematic mapping and survey. *Computers & Security*, 108:102366, 2021.

Lu Jinping, Tang Zhi, Mao Jian, Gu Zhiling, and Zhang Jiemin. Mixed-models method based on machine learning in detecting webshell attack. In *Proceedings of the 2020 International Conference on Computers, Information Processing and Advanced Education*, pp. 251–259, 2020.

Inwon Kang, Sikai Ruan, Tyler Ho, Jui-Chien Lin, Farhad Mohsin, Oshani Seneviratne, and Lirong Xia. Llm-augmented preference learning from natural language. *arXiv preprint arXiv:2310.08523*, 2023.

Ha V Le, Tu N Nguyen, Hoa N Nguyen, and Linh Le. An efficient hybrid webshell detection method for webserver of marine transportation systems. *IEEE Transactions on Intelligent Transportation Systems*, 24(2):2630–2642, 2021.

Hongfu Liu and Ye Wang. Towards informative few-shot prompt with maximum information gain for in-context learning. *arXiv preprint arXiv:2310.08923*, 2023.

Lei Liu and Ziyou He. Webshell detection based on codebert and deep learning. In *Proceedings of the 2023 13th International Conference on Communication and Network Security*, pp. 32–39, 2023.

Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. On the reliability and explainability of language models for program generation. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–26, 2024.

Mingrui Ma, Lansheng Han, and Chunjie Zhou. Large language models are few-shot generators: Proposing hybrid prompt algorithm to generate webshell escape samples. *arXiv preprint arXiv:2402.07408*, 2024.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.

Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*, 2024.

Christos Petridis. Text classification: Neural networks vs machine learning models vs pre-trained models. *arXiv preprint arXiv:2412.21022*, 2024.

Ao Pu, Xia Feng, Yuhan Zhang, Xuelin Wan, Jiaxuan Han, and Cheng Huang. Bert-embedding-based jsp webshell detection on bytecode level using xgboost. *Security and Communication Networks*, 2022(1):4315829, 2022.

Longchen Qi, Rui Kong, Yang Lu, and Honglin Zhuang. An end-to-end detection method for webshell with deep learning. In *2018 Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC)*, pp. 660–665. IEEE, 2018.

Leonardo Rigutini, Achille Globo, Marco Stefanelli, Andrea Zugarini, Sinan Gultekin, Marco Ernandes, et al. Performance, energy consumption and costs: a comparative analysis of automatic text classification approaches in the legal domain. *INTERNATIONAL JOURNAL ON NATURAL LANGUAGE COMPUTING*, 13(1):19–35, 2024.

Mengchuan Shang, Xueying Han, Changzhi Zhao, Zelin Cui, Dan Du, and Bo Jiang. Multi-language webshell detection based on abstract syntax tree and treelstm. In *2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 377–382. IEEE, 2024.

Oleksii Starov, Johannes Dahse, Syed Sharique Ahmad, Thorsten Holz, and Nick Nikiforakis. No honor among thieves: A large-scale analysis of malicious web shells. In *Proceedings of the 25th International Conference on World Wide Web*, pp. 1021–1032, 2016.

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

Cisco Talos. Talos ir trends q4 2024. https://blog.talosintelligence.com/talos-ir-trends-q4-2024/, 2024.

Fouad Trad and Ali Chehab. Manual prompt engineering is not dead: A case study on large language models for code vulnerability detection with dspy. In *2025 8th International Conference on Data Science and Machine Learning Applications (CDMA)*, pp. 168–173. IEEE, 2025.

Guan-Yu Wang, Hung-Jui Ko, Chang-Po Chiang, and Wei-Jen Wang. Webshell detection based on codebert and deep learning model. In *Proceedings of the 2024 5th International Conference on Computing, Networks and Internet of Things*, pp. 484–489, 2024a.

Yanlin Wang, Yanxian Huang, Daya Guo, Hongyu Zhang, and Zibin Zheng. Sparsecoder: Identifier-aware sparse transformer for file-level code summarization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 614–625. IEEE, 2024b.

Zhiqiang Wang, Haoyu Wang, and Lu Hao. Poster: Long php webshell files detection based on sliding window attention. *arXiv preprint arXiv:2502.19257*, 2025.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Peiwen Yuan, Shaoxiong Feng, Yiwei Li, Xinglin Wang, Yueqi Zhang, Chuyi Tan, Boyuan Pan, Heda Wang, Yao Hu, and Kan Li. Focused large language models are stable many-shot learners. *arXiv preprint arXiv:2408.13987*, 2024.

Qing-peng ZENG, Jiang-li CHAI, and Jian-sheng WU. A webshell detection method based on multi-feature fusion and federated learning. 2025.

Yifan Zhang, Haiyan Kang, and Qiang Wang. Mmfdetect: Webshell evasion detect method based on multimodal feature fusion. *Electronics*, 14(3):416, 2025.

A Prompt Details

Prompt for WebShell Detection

System Prompt: You are tasked with analyzing PHP scripts. Your objective is to classify the provided PHP code as either a webshell or a legitimate script. A webshell is typically a malicious script intended to exploit the server by executing unauthorized commands or providing backdoor access.
User Prompt: Analyze the provided PHP code to determine whether it constitutes a webshell or a legitimate script. Provide your verdict as webshell or benign.
[Critical Code]
[Source Code]
[Examples]
Output:

B Critical Function Details

Table 1: Statistics of Critical Functions in Webshell and Benign Programs. This table reports the percentage of files containing each function category and the average number of occurrences per file, with the "Total" row aggregating statistics across all categories.

Function Category	Metric	Webshell Files	Normal Files
Program Execution	Files with Function (%)	53.06	1.54
	Avg. Occurrences per File	3.21	0.03
Code Execution	Files with Function (%)	85.03	14.79
	Avg. Occurrences per File	8.30	0.36
Callback Functions	Files with Function (%)	34.69	6.47
	Avg. Occurrences per File	0.92	0.11
Network Communication	Files with Function (%)	50.34	2.77
	Avg. Occurrences per File	1.69	0.04
Information Gathering	Files with Function (%)	46.26	2.77
	Avg. Occurrences per File	5.46	0.05
Obfuscation and Encryption	Files with Function (%)	69.39	9.86
	Avg. Occurrences per File	3.19	0.16
Total (All Functions)	Files with Function (%)	91.16	20.49
	Avg. Occurrences per File	22.76	0.74

C Dataset Details

Table 2: Dataset Composition, Distribution, and Sources. The dataset comprises 26,594 PHP scripts, categorized into benign programs and webshells, with their respective counts, proportions, and sources.

Category	Count	Percentage	Source References
Benign Programs	21,665	81.5%	Grav, OctoberCMS, Laravel, WordPress, Joomla, Nextcloud, Symfony, CodeIgniter, Yii2, CakePHP, Intervention/Image, Typecho
Webshells	4,929	18.5%	WebShell, WebshellSample, Awsome-Webshell, PHP-Bypass-Collection, Web-Shell (tdifg), Webshell (lhlsec), PHP-Backdoors, Tennc/Webshell, PHP-Webshells, BlackArch/Webshells, Webshell-Samples, Programe, WebshellDetection, WebShell-Collection, PHP-Backdoors (1337r0j4n), PHP-Webshell-Dataset, Xiao-Webshell
Total	26,594	100.0%	—

Table 3: Normalized Scores for Key Function Categories. These scores reflect the weighted behavioral significance of each category as computed by the WBFP method.

Function Category	Normalized Score
Program Execution	0.2068
Code Execution	0.2081
Callback Functions	0.0790
Network Communication	0.1498
Information Gathering	0.1861
Obfuscation and Encryption	0.1702

D Results

Table 4: Performance Comparison of BFAD-Enhanced Models Against Baselines. This table compares traditional ML and DL models with large and small-scale LLMs, both standalone and enhanced with BFAD.

Category	Model	Accuracy	Precision	Recall	F1 Score
Sequence Baselines	GloVe+SVM	96.20%	93.30%	94.30%	93.80%
	CodeBERT+RF	96.30%	94.00%	95.60%	94.80%
Graph Baselines	GCN	96.90%	94.40%	95.30%	94.90%
	GAT	98.37%	99.52%	97.39%	98.87%
LLM Baselines (Large)	GPT-4	97.27%	100.00%	85.98%	92.46%
	LLaMA-3.1-70B	98.01%	97.31%	92.36%	94.77%
	Qwen-2.5-Coder-14B	98.64%	99.32%	93.63%	96.39%
LLM Baselines (Small)	Qwen-2.5-Coder-3B	71.11%	38.93%	99.32%	55.93%
	Qwen-2.5-3B	93.72%	78.03%	91.84%	84.37%
	Qwen-2.5-1.5B	43.62%	34.61%	95.77%	50.84%
	Qwen-2.5-0.5B	19.47%	18.65%	100.00%	31.44%
LLM + BFAD	GPT-4	99.75%	100.00%	98.71%	99.35% (+6.89)
	LLaMA-3.1-70B	99.38%	98.72%	98.09%	98.40% (+3.63)
	Qwen-2.5-Coder-14B	98.76%	98.68%	94.90%	96.75% (+0.36)
	Qwen-2.5-Coder-3B	78.89%	46.67%	100.00%	63.64% (+7.71)
	Qwen-2.5-3B	97.39%	88.64%	99.36%	93.69% (+9.32)
	Qwen-2.5-1.5B	80.40%	48.51%	100.00%	65.33% (+14.49)
	Qwen-2.5-0.5B	91.94%	71.10%	98.73%	82.67% (+51.23)

Table 5: Performance of Context-Aware Code Extraction with Qwen-2.5-3B with Different Context Lengths and Strategies.

Method	Accuracy	Precision	Recall	F1 Score
Source Code (Vanilla)	93.72%	78.03%	91.84%	84.37%
Critical Regions ($\tau = 100$)	96.28%	86.71%	95.54%	90.91%
Critical Regions ($\tau = 200$)	95.78%	84.75%	95.54%	89.82%
Critical Regions ($\tau = 300$)	95.04%	82.32%	94.90%	88.17%
Hybrid Strategy ($\tau = 100$)	**96.40**%	**89.02**%	**92.99**%	**90.97**%
Hybrid Strategy ($\tau = 200$)	95.66%	85.47%	93.63%	89.36%
Hybrid Strategy ($\tau = 300$)	95.78%	85.55%	94.27%	89.70%

Table 6: Performance of Context-Aware Code Extraction with GPT-4 with Different Context Lengths and Strategies.

Method	Accuracy	Precision	Recall	F1 Score
Source Code (Vanilla)	97.27%	100.00%	85.98%	92.46%
Critical Regions ($\tau = 100$)	98.51%	99.32%	92.99%	96.05%
Critical Regions ($\tau = 200$)	99.01%	99.34%	95.54%	97.40%
Critical Regions ($\tau = 300$)	99.01%	98.69%	96.18%	97.42%
Hybrid Strategy ($\tau = 100$)	99.01%	100.00%	94.90%	97.39%
Hybrid Strategy ($\tau = 200$)	99.14%	100.00%	95.81%	97.86%
Hybrid Strategy ($\tau = 300$)	**99.38**%	**100.00**%	**96.82**%	**98.38**%

Table 7: Comparison of Demonstration Selection Strategies for In-Context Learning with Qwen-2.5-3B (Under Best Hybrid Strategy $\tau = 100$).

Method	**Accuracy**	**Precision**	**Recall**	**F1 Score**
No-ICL	93.72%	78.03%	91.84%	84.37%
Random	77.79%	46.33%	88.53%	60.83%
SC-Sim	93.42%	78.57%	91.08%	84.36%
WBFP-Eq	96.98%	86.39%	99.32%	92.41%
WBFP-Wt	**97.39%**	**88.64%**	**99.36%**	**93.69%**

Table 8: Comparison of Demonstration Selection Strategies for In-Context Learning with GPT-4 (Under Best Hybrid Strategy $\tau = 300$).

Method	**Accuracy**	**Precision**	**Recall**	**F1 Score**
No-ICL	99.38%	100.00%	96.82%	98.38%
Random	89.00%	65.58%	90.97%	76.22%
SC-Sim	98.63%	100.00%	92.90%	96.32%
WBFP-Eq	99.62%	100.00%	98.03%	99.01%
WBFP-Wt	**99.75%**	**100.00%**	**98.71%**	**99.35%**