PART 1: Failure Cases and Deeper Analysis

Tested demo 1: Benign Programs

WebShell Demo1

LLMs Output:

{
"WebshellDetectDataset/php/white/1986.php.txt": {
"model": "qwen2.5-0.5b-instruct",
"response": "**Webshell (Malicious)**\n\nThe given PHP code snippet is designed to be a webshell. It contains functions for encoding data into Base64 format and decoding Base64 data back into plain text. This type of code is commonly used to execute arbitrary commands or inject malicious scripts onto websites.",
}
}

{
"WebshellDetectDataset/php/white/1986.php.txt": {
"model": "qwen2.5-1.5b-instruct",
"response": "**Webshell (Malicious)**\n\n### Explanation:\n\nThe provided PHP code appears to be part of a custom implementation of Base64 encoding/decoding functions using Sodium's crypto primitives. This kind of functionality could potentially be used by attackers to encrypt sensitive data or manipulate URLs within a PHP application.\n\nHowever, without more context about its use within a specific PHP environment or any suspicious activities, this alone does not definitively make it a webshell.",
}
}

{
"WebshellDetectDataset/php/white/1986.php.txt": {
"model": "llama3.1-70b-instruct",
"response": "**Normal**\n\nThis PHP code is a part of the Paragon Initiative Enterprises' Sodium library, which is a cryptography library for PHP. The code defines an abstract class `ParagonIE_Sodium_Core_Base64_Common` that provides methods for encoding and decoding base64 strings. The class is designed to work with different

character sets and provides options for padding and strict padding.\n\nThe code is well-structured, follows object-oriented programming principles, and includes documentation comments. It does not contain any malicious code or suspicious",
  }
}

Tested demo 2: WebShell

WebShell Demo2

LLMs Output:

{
  "WebshellDetectDataset/php/black/22750.php.txt": {
    "model": "qwen2.5-1.5b-instruct",
    "response": "****Malicious Webshell****\n\nThis PHP code appears to be a webshell because:\n\n1. It includes numerous `eval()` calls, which are used for executing arbitrary code.\n2. It uses the `fopen`, `fwrite`, and `fclose` functions to manipulate files, suggesting it has access to write capabilities.\n3. The use of `eval` allows the attacker to execute potentially dangerous scripts, which can compromise the server or steal sensitive information.\n4. It seems to have functionality that could",
  }
}

{
  "WebshellDetectDataset/php/black/22750.php.txt": {
    "model": "qwen2.5-3b-instruct",
    "response": "**Webshell**\n\nThe provided PHP code appears to be a functional webshell. It allows users to select files, directories, and other resources to include in a ZIP archive and then compress them into a specified output file. The script includes features like file selection, directory traversal, and execution of arbitrary commands within the context of the server where it resides. These capabilities are typical of a webshell, which is used by attackers to gain unauthorized access and control over a web server.",

{
  "WebshellDetectDataset/php/black/22750.php.txt": {
    "model": "llama3.1-70b-instruct",

"response": "**Normal**\n\nThis PHP script is a simple file archiver that allows users to select files and directories to be zipped and downloaded. It does not contain any malicious code or functionality that would classify it as a webshell. The script uses a custom PHPzip class to handle the zipping process, and it does not have any capabilities to execute system commands, access sensitive data, or perform other malicious actions.",
  }
}

Why Different Scales of WebShell Behave Differently

The behavior of WebShell detection varies across LLMs of different sizes due to their differing capacities for reasoning, pattern recognition, and contextual understanding. This is evident when comparing smaller models (e.g., Qwen-0.5B, Qwen-1.5B) to larger ones (e.g., LLaMA 3.1-70B).

1. Smaller LLMs (e.g., Qwen-0.5B, Qwen-1.5B)

- **Behavior**: Smaller models rely heavily on **surface-level pattern matching**. They identify code as malicious based on specific keywords or functions (e.g., base64_decode, eval()) without fully grasping the broader context.↳
- **Example from Demo 1 (Benign Code)**:
  - **Qwen-0.5B**: Misclassifies a benign cryptography library as malicious, citing the presence of base64_decode as suspicious, despite its legitimate use.
  - **Qwen-1.5B**: Also labels it malicious but shows slight improvement by noting the Sodium library context, though it still defaults to a malicious classification.
- **Reason**: With fewer layers and attention heads, smaller models lack the depth to analyze beyond superficial indicators, leading to overgeneralization and frequent misclassification of benign code.↳

2. Larger LLMs (e.g., LLaMA 3.1-70B)

- **Behavior**: Larger models excel at **contextual reasoning**, evaluating the overall structure, purpose, and intent of the code rather than focusing solely on individual functions.
- **Example from Demo 1 (Benign Code)**:

    - **LLaMA 3.1-70B**: Correctly identifies the benign code as part of the Sodium cryptography library, recognizing its well-structured, non-malicious nature.
- **Example from Demo 2 (Malicious Code)**:
    - **LLaMA 3.1-70B**: Incorrectly classifies a malicious WebShell as normal, interpreting it as a "simple file archiver" and overlooking features like arbitrary command execution.
- **Reason**: With more layers and attention heads, larger models capture complex patterns and semantics. However, this broader focus can lead them to miss specific malicious indicators if the code is disguised or lacks overtly suspicious traits.

Key Differences

- **Smaller Models**: Depend on shallow heuristics, resulting in **lower precision** (frequent false positives) as they flag benign code based on isolated features.
- **Larger Models**: Leverage deeper understanding, but this can reduce **recall** (more false negatives) if they fail to detect well-obfuscated threats.

These differences likely stem from model architecture: smaller models prioritize surface patterns due to limited capacity, while larger models attend to deeper features but may overgeneralize or be overly lenient.

Why Errors Occur in Detection

Errors in WebShell detection arise from the inherent limitations of LLMs at different scales, as demonstrated in the examples:

1. False Positives in Smaller Models

- **Cause**: Their shallow reasoning misinterprets legitimate functions as malicious due to a lack of contextual awareness.
- **Example**: In Demo 1, Qwen-0.5B flags `base64_decode` in a benign library as a WebShell indicator, ignoring its cryptographic purpose.
- **Impact**: High rate of benign code misclassification, reducing precision.

2. False Negatives in Larger Models

- **Cause**: Over-reliance on broad context can cause them to overlook specific malicious behaviors, especially in disguised or subtle WebShells.
- **Example**: In Demo 2, LLaMA 3.1-70B misses the malicious intent of a WebShell, focusing on its surface-level "file archiver" appearance rather than its command execution capabilities.
- **Impact**: Failure to detect threats, lowering recall.

Trade-Off

Smaller models are overly cautious but inaccurate, while larger models are more discerning but risk missing threats. This trade-off reflects their differing abilities to balance sensitivity and specificity.

Mitigation with the BFAD Framework

The proposed **BFAD framework** addresses these issues by:

- **Enhancing Smaller Models**: Guides them to focus on function-level behaviors rather than superficial patterns, reducing false positives.

- **Refining Larger Models**: Directs their attention to critical functions and indicators, minimizing false negatives.

This dual approach improves detection accuracy across scales, as noted in your response to feedback, by balancing precision and recall.

Part 2: Limitations and Future Directions

Given that LLMs have not yet been widely applied to WebShell detection, we summarize current limitations and outline potential future research directions. Our goal is to help advance this emerging field and inspire more comprehensive exploration.

1. Dataset Limitations and Research Opportunities

As discussed in our response to *Reviewer 1h5H Q2*, WebShell detection is fundamentally constrained by privacy concerns and the scarcity of real-world data. Unlike other code-related tasks, such as vulnerability detection, nearly all prior studies rely on a limited set of publicly available, open-sourced WebShell datasets. This makes it difficult to evaluate how well detection methods generalize to novel or obfuscated threats.

A key open question is how to simulate and evaluate the effectiveness of detection models against diverse attack types **without compromising privacy**. One promising direction is to **use LLMs to generate synthetic, class-specific WebShell variants**. These LLM-generated samples could serve as a privacy-preserving benchmark suite for stress-testing detection systems under a wider range of threat scenarios.

2. LLM Limitations and Research Opportunities

Even with advanced language models, several technical challenges remain before such systems can be reliably deployed in real-world WebShell detection pipelines:

- **Obfuscation and evasion**: Novel WebShells may obfuscate or hide malicious function calls, evading static analysis.

- **Zero-day threats**: Attackers could use new, previously unseen functions that are not covered by existing rulebases.

To address these challenges, we propose several research directions:

- **Incorporating dynamic analysis**: Recent work shows that analyzing runtime function call traces (e.g., captured via sandbox execution) provides valuable behavioral signals for detecting WebShells. Integrating such dynamic data can help uncover hidden or obfuscated behavior, and enhance the effectiveness of our *Critical Function Filter* module.

- **Leveraging reasoning-capable models ("fast + slow thinking")**: In real-world settings, we envision a two-stage architecture: a fast first-stage LLM performs lightweight filtering using domain knowledge, followed by a slower, reasoning-oriented model (e.g., DeepSeek-R1, GPT-o3) that deeply analyzes ambiguous samples. While this multi-stage pipeline was not evaluated in the current study due to data constraints, it represents a promising path forward for adaptive threat detection.

- **Dynamic rule expansion via agent-based workflows**: We propose embedding the rule update process into an autonomous agent loop, where LLMs assist in identifying and incorporating new malicious patterns into the detection system. This supports continuous adaptation in response to evolving threats.

- **Fine-tuning domain-specific LLMs**: Our analysis reveals that both small and large LLMs face bottlenecks in understanding complex code semantics. Our paper shows that **domain-specific fine-tuning**—especially on models already optimized for code understanding (e.g., Qwen-Code)—can significantly improve detection accuracy. Since our study demonstrates the viability of using LLMs in this domain, we are optimistic that fine-tuned models on the WebShell dataset will yield even better performance.

We think these directions represent important steps toward making LLM-based WebShell detection more robust, interpretable, and adaptable to real-world adversarial settings.