Beyond Detection: A Comprehensive Benchmark and Study on Representation Learning for Fine-Grained Webshell Family Classification

Feijiang Han
University of Pennsylvania
feijhan@seas.upenn.edu

Abstract

Malicious WebShells represent a severe and evolving threat, compromising critical digital infrastructures and endangering public services in sectors such as healthcare and finance. While the research community has achieved considerable success in WebShell detection (distinguishing malicious from benign samples), we argue it is time to advance from passive detection to a new stage of in-depth analysis and proactive defense. A promising and critical direction is the automation of WebShell family classification: identifying the specific malware lineage to understand an adversary's tactics and enable a precise, rapid response. This crucial task, however, remains a largely unexplored area that currently relies on slow, manual expert analysis. To address this gap, we present the first systematic study to automate WebShell family classification. Our method begins with extracting dynamic function call traces to capture inherent behaviors that are resistant to common encryption and obfuscation. To enhance the scale and diversity of our dataset for a more stable evaluation, we augment these real-world traces with new variants synthesized by a Large Language Model (LLM). These augmented traces are then abstracted into sequences, graphs, and trees, providing a foundation to benchmark a comprehensive suite of representation methods. Our evaluation spans classic sequence-based embeddings (CBOW, GloVe), transformers (BERT, SimCSE), and a range of structure-aware algorithms, including Graph Kernels, Graph Edit Distance, Graph2Vec, and various Graph Neural Networks. Through extensive experiments on four real-world, family-annotated datasets under both supervised and unsupervised settings, we establish a robust baseline and provide practical insights into the most effective combinations of data abstractions, representation techniques, and learning paradigms for this challenge. This foundational work is a crucial step toward automating threat intelligence, accelerating incident response, and ultimately enhancing the resilience of the digital services that society depends on.

1 Introduction

Malicious WebShells have evolved from simple scripts into strategic assets used in sophisticated attacks that directly threaten critical public services in sectors like healthcare and finance, endangering the sensitive data of millions. To counter this pervasive threat, the research community has achieved considerable success in developing automated techniques for WebShell detection (Tu et al. 2014; Aboaoja et al. 2022; Ma, Han, and Zhou 2024; Feng et al. 2024; Han et al. 2025).

While successful, this focus on binary classification (malicious vs. benign) provides only a foundational first line of defense and offers limited actionable intelligence for subsequent security operations. A more proactive and robust security posture requires not just knowing that a server is compromised, but understanding the specific nature of the threat itself. This necessitates **WebShell family classification**, the task of identifying the specific variant or lineage of the malware. Automating this process is crucial as it unlocks a deeper level of threat intelligence, helping security teams attribute attacks, anticipate an adversary's next moves, and mount a faster, more targeted incident response (Zhao et al. 2024). For instance, an automated system can reduce response time from hours of manual expert analysis to mere seconds, enabling security operation centers (SOCs) to trigger specific defense playbooks tailored to a family's known tactics before significant damage, like data exfiltration, occurs. This critical task, however, remains largely unexplored in the research community, with current practices relying on time-consuming manual analysis.

We argue that automating this task is technically feasible for two primary reasons. First, WebShells within the same family often share distinct behavioral characteristics due to code reuse (Wrench and Irwin 2015; Starov et al. 2016). Second, this malicious behavior can be captured in the program's dynamic function call traces even when the source code is obfuscated (De Goër et al. 2018; Xu and Chen 2023). If a model can learn to recognize these behavioral similarities, it can effectively group and track WebShell families.

However, family classification is inherently more challenging than binary detection, as it requires models that can capture the nuanced behavioral patterns that differentiate families, not just generic malicious traits. This challenge motivates the foundational research question of our work: ***What data structures and representation methods are most effective for capturing these family-specific behaviors?***

To answer this question, this paper presents the first systematic study of WebShell family classification by performing a large-scale empirical evaluation of various representations and encoding methods derived from WebShell behaviors. Our goal is to establish a robust foundation and a practical guide for this critical task.

Our contributions are as follows:

- **A Comprehensive Methodological Framework.** We design and execute the first large-scale benchmark for this task. To ensure a robust evaluation, we introduce a data synthesis framework leveraging a Large Language Model (LLM) to augment our real-world data with diverse, behaviorally-consistent function call traces. We abstract this enriched dataset into three fundamental data structures (sequences, graphs, and trees) and systematically evaluate a diverse spectrum of representation learning methods, from classic embeddings and transformers to structure-aware algorithms like Graph Kernels and various Graph Neural Networks (GNNs).
- **A Robust Empirical Baseline.** Through extensive experiments on four real-world and LLM-augmented datasets with both supervised and unsupervised classification, we establish the first robust, data-driven performance baseline for WebShell family classification. This provides a crucial point of comparison for all future work in this emerging area.
- **Actionable Insights for the Security Community.** Our analysis delivers a clear hierarchy of performance, demonstrating that structural representations (especially trees) are decisively more effective than sequential ones, and that GNNs are the premier modeling architecture. These findings offer immediate, practical guidance for practitioners and researchers aiming to build effective classification systems.
- **A Practical Guide to Implementation.** We distill our findings into a set of best practices for implementation, detailing optimal strategies for model selection and hyperparameter configuration based on the downstream task. This guide empowers even under-resourced organizations to deploy more advanced defensive tools.

Ultimately, this work provides both a foundational benchmark and a practical guide, empowering the security community to move beyond simple detection and build the next generation of intelligent, fine-grained defense systems.

2 Problem Formulation

The primary goal of WebShell family classification is to automatically categorize a given malicious WebShell into one of several predefined families. Our central research objective is to fundamentally understand which data abstractions and representation methods are most effective at capturing these family-specific features from raw data. We therefore adopt a two-stage framework that decouples representation learning from classification, enabling a fair and standardized benchmark of different encoders.

Stage 1: Representation Learning. The input to this stage is a raw, unstructured function call trace from a single WebShell. An encoder model, g, maps this trace into a fixed-dimensional numerical vector $\mathbf{x} \in \mathbb{R}^d$:

$$\mathbf{x} = g(\text{trace}) \tag{1}$$

This vector $\mathbf{x}$, or embedding, is a structured summary of the WebShell's runtime behavior. Our core investigation lies in comparing various designs for this encoder g.

Stage 2: Benchmarking via Classification. In this stage, we use a suite of standard classifiers to benchmark the quality of the embeddings ($\mathbf{x}$) produced in Stage 1. The central principle is that a higher-quality representation will be more separable in vector space and thus yield better performance on downstream tasks, providing an objective measure of the encoder's effectiveness. Formally, the classification task is defined as follows: given a dataset of embeddings $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, where $y_i \in \{1, \ldots, K\}$ is the family label for one of K families. The objective is to learn a classifier $f : \mathbb{R}^d \to \{1, \ldots, K\}$ that accurately predicts the label $\hat{y} = f(\mathbf{x})$ for any given embedding $\mathbf{x}$.

3 Dataset Collection

Data Acquisition and Annotation. Our dataset construction follows an established pipeline for creating high-quality, real-world WebShell datasets (Zhao et al. 2024). The process begins with suspicious files flagged by a large-scale cloud provider's malware detection system. Each potential WebShell is executed in a secure sandbox to capture its dynamic function call trace—a chronologically ordered log of its runtime behavior. This dynamic approach offers a significant advantage over static analysis, bypassing common evasion techniques like obfuscation and encryption, thereby revealing a well-defined structure of operational behaviors ideal for extracting family-specific features. An example of this raw data is shown in Table 1.

Security experts then manually review these traces to filter out false positives. The verified malicious samples subsequently undergo a human-machine collaborative process for family annotation. To ensure high reliability, each family label is confirmed by at least two independent experts. Samples that cannot be confidently assigned to any established family are designated as outliers and assigned a 'Family ID' of -1. The final labeled data format is presented in Table C.1 in Appendix C.

LLM-Powered Data Augmentation. To address the inherent limitations of real-world data collection, such as data scarcity for rare families and the absence of novel, zero-day threats, we introduce an LLM-augmented data synthesis framework to enrich and expand our dataset. This framework enables us to scale our data collection efforts and enhance the diversity of the samples.

Specifically, we employed a two-pronged strategy. First, for **Intra-Family Data Augmentation**, we used a few-shot prompting technique, providing the LLM with a high-level description of a family's behavior along with several canonical examples of its function call traces. This enabled the model to generate a large volume of new samples that are behaviorally consistent with existing families but syntactically unique. This technique effectively addresses the class imbalance problem by augmenting underrepresented families. The specific prompt templates used in this process are detailed in Appendix B.

Second, building on this, we introduced a **New Family & Zero-Day Simulation**. This stage simulates the adversarial tactic of creating novel variants by blending the behavioral characteristics of different malware families. The resulting

Table 1: Examples of raw dynamic function call sequences captured from sandboxed execution. Each record consists of a unique identifier ('Filemd5') and the corresponding list of function calls.

Filemd5	Dynamic Function Calls
8ab45f6a5901e607794bb6d846b33b01	["_main_", "zend_compile_file", "_main_", "base64_decode", "_main_", "assert", "assert", "zend_compile_string", "assert", "zend_fetch_r_post", "assert", "eval", "eval", "zend_compile_string", ...]
191c2b18cf75e72194dff925ea34bd3d	["_main_", "zend_compile_file", "_main_", "set_time_limit", "_main_", "header", "_main_", "zend_fetch_r_server", "_main_", "getHTTPPage", "getHTTPPage", "stream_context_create", "getHTTPPage", "file_get_contents", ...]

synthetic traces can be labeled either as entirely new families or as adversarial outliers, which are designed to challenge the classifier's robustness and evaluate its zero-shot learning capabilities.

To ensure the fidelity and logical soundness of the synthetic data, all LLM-generated traces underwent a final verification and sanitization phase.

Dataset Details. Through this process, we constructed four distinct datasets for our experiments, labeled DS1 through DS4. These datasets feature progressively increasing scale and complexity in terms of sample size, the number of families, and the quantity of outliers. This graduated design allows for a robust and thorough evaluation of our representation methods across diverse conditions. Table D.1 in Appendix D summarizes the key statistics of each dataset.

4 Behavioral Data Abstraction

To make raw function call traces amenable to machine learning, we abstract this sequential data into three distinct structural representations: sequences, graphs, and trees. As illustrated in Figure 1, each representation captures a different aspect of a WebShell's runtime behavior, providing a unique lens through which to analyze its characteristics.

Sequence Model. The most direct abstraction treats a function call trace as a sequence of discrete tokens, where each function name becomes a token in the execution order (Figure 1a). This linear representation is compatible with a wide range of natural language processing models. A trace can be represented as $S = (t_1, t_2, \ldots, t_n)$, where t_i is the i-th function called.

Graph Model. To capture more complex, non-sequential interactions, we model each trace as a Function Call Graph (FCG), shown in Figure 1b. An FCG, $G = (V, E)$, provides a static, aggregate view of the program's behavior, where each unique function is a node $v \in V$, and a directed edge $(u, v) \in E$ exists if function u ever calls function v. The edges can be weighted by call frequency to represent the strength of the interaction. This model effectively captures all calling relationships, including loops and indirect calls.

Tree Model. To preserve the hierarchical nature of program execution, we also represent each trace as a Function Call Tree (FCT), illustrated in Figure 1c. The FCT, $T = (V, E)$, is a rooted tree where the entry point (e.g., '_main_') is the root and edges represent direct parent-child call relationships. Unlike the graph model, the FCT is acyclic and preserves the specific execution path and context; a function called multiple times in different contexts appears as distinct nodes in the tree.

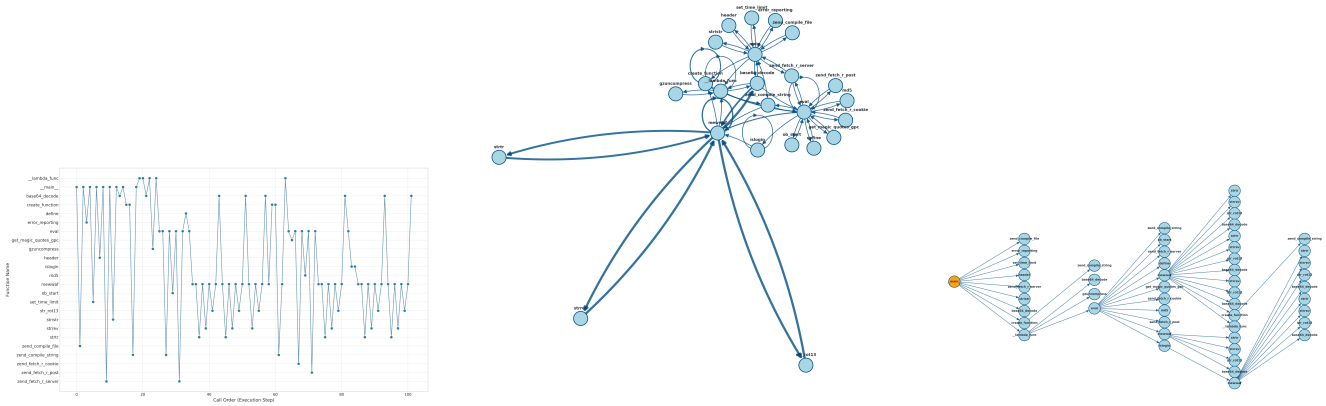
5 Representation and Benchmarking

5.1 Representation Learning Methods

We apply a diverse set of foundational and widely adopted representation learning techniques tailored to each data abstraction. Table 2 provides a complete overview.

For sequence models, we evaluate two distinct categories: classic context-free methods (CBOW (Mikolov et al. 2013), GloVe (Pennington, Socher, and Manning 2014)) and modern context-aware transformers (BERT (Devlin et al. 2018), SimCSE (Gao, Yao, and Chen 2021)). To produce a single fixed-dimensional vector for each function call trace, we employ several aggregation strategies tailored to each model type. For the static embeddings produced by CBOW and GloVe, we investigate three strategies: averaging all function call vectors to create a mean representation (avg); concatenating the vectors of each function call sequentially (concat); and a TF-IDF weighted average that emphasizes more discriminative functions. For the transformer models, we leverage their deep, contextualized hidden states by: averaging the hidden states across all tokens (avg); concatenating the hidden state vectors from different layers (concat); and using the final hidden state of the dedicated classification token, [CLS].

For graph and tree models, we employ two classic methods for direct structural comparison. First, Graph/Tree Kernels (Shervashidze et al. 2011) measure similarity by counting shared substructures, such as common call sequences (paths), randomly generated traversals (random walks), and identical small-scale call hierarchies (subtrees). Second, we



(a) Sequence Model (b) Graph Model (c) Tree Model

Figure 1: The visualization of three data abstractions. (a) The Sequence Model visualizes the chronological execution flow. (b) The Graph Model provides a static, aggregate view of all calling relationships. (c) The Tree Model preserves the hierarchical call structure and execution context.

compute the Graph/Tree Edit Distance (Marzal and Vidal 1993), which quantifies dissimilarity by calculating the minimum cost of operations (e.g., node insertion, deletion, and substitution) required to transform one structure into another. For learning-based approaches, we benchmark three prominent Graph Neural Network (GNN) architectures which learn representations via message passing: Graph Convolutional Network (GCN) (Kipf and Welling 2016), Graph Attention Network (GAT) (Veličković et al. 2018), and Graph Isomorphism Network (GIN) (Xu et al. 2018). Finally, we include Graph2Vec (Narayanan et al. 2017), an unsupervised method that learns whole-graph embeddings.

Table 2: Overview of the representation methods and their implementation variants evaluated for each data abstraction.

Representation Method	Implementation Variants
Sequence-Based Models	
Word2Vec (CBOW)	concat, avg, concat & avg
GloVe	concat, avg, concat & avg
BERT	concat, avg, cls
SimCSE	concat, avg, cls
Graph-Based Models	
Graph Kernel	Path, Walk, Subtree
Graph Edit Distance	–
Graph Neural Networks	GCN, GAT, GIN
Graph Embedding (Graph2Vec)	–
Tree-Based Models	
Tree Kernel	Path, Walk, Subtree
Tree Edit Distance	–
Graph Neural Networks	GCN, GAT, GIN
Tree Embedding (Graph2Vec)	–

Benchmarking Classifiers. To assess the quality of the learned representations, we benchmark the resulting embeddings using a suite of four standard classifiers. For unsupervised evaluation, we use K-Means (MacQueen 1967) and Mean-Shift (Comaniciu and Meer 2002) clustering. For supervised evaluation, we employ two widely-used models: Random Forest (Breiman 2001) and the Support Vector Machine (SVM) (Cortes and Vapnik 1995). This comprehensive framework allows us to measure the effectiveness of each representation in both labeled and unlabeled settings.

6 Experimental Setup

6.1 Implementation Details

Representation Models. For all representation learning models, we standardized the output embedding dimension to 128 to balance between expressiveness and computational efficiency. The input dimensions were dynamically set based on the function vocabulary size of each specific dataset. To establish a consistent and reproducible baseline, we utilized the default hyperparameter settings (e.g., optimizer, learning rate, loss function) recommended for each model during the representation learning phase. The detailed configurations are provided in Appendix A.

Benchmarking Classifiers. To ensure a fair comparison across different representations, we employed a grid search with cross-validation to identify the best-performing parameter configuration for each representation-classifier pair. To ensure statistical robustness, all reported results are the average of 10 independent runs, each with a different seed.

6.2 Evaluation Metrics

Supervised Classification. We assess the performance of supervised models using four standard metrics: Accuracy, Precision, Recall, and F1-score. For the multi-class setting, Precision, Recall, and F1-score are reported as macro-averaged values. This approach computes the metric in-

dependently for each family and then calculates the unweighted mean, ensuring that all families, regardless of their size, contribute equally to the final score.

Unsupervised Clustering. We evaluate clustering quality using two primary metrics: Accuracy and Normalized Mutual Information (NMI). Accuracy is computed by first finding the optimal mapping between cluster assignments and ground-truth labels via the Hungarian algorithm and then calculating the percentage of correctly assigned samples. NMI measures the agreement between the assigned clusters C and true labels Y, correcting for chance:

$$\text{NMI}(Y, C) = \frac{2 \times I(Y; C)}{H(Y) + H(C)}, \tag{2}$$

where $I(Y; C)$ is the mutual information between the true and predicted labels, while $H(Y)$ and $H(C)$ are their respective entropies.

7 Results and Analysis

We present our experimental findings, focusing on the more complex **DS3 and DS4 datasets** (full results in Figures 2 and 3; DS1/DS2 in Appendix E). Performance is visualized using a blue-to-red color gradient, where blue signifies higher scores. The top three results in each column are highlighted in bold.

7.1 Key Insight 1: Structural Semantics Definitively Outperform Sequential Syntax

The most striking result from our benchmark is the significant performance gap between structural (graph and tree) and sequential representations. As shown in the result figures, GNNs and even classic methods like Tree Edit Distance consistently achieve F1-scores exceeding 0.9, while the performance of advanced sequence models like BERT is both lower and more volatile. Notably, as dataset complexity increases, structural methods exhibit a much more graceful performance degradation, reinforcing their robustness and scalability for this task.

This finding points to a fundamental limitation of sequential models. They are designed to capture linear dependencies, treating a function call trace as a syntactic sentence. However, a WebShell family's signature lies not in call adjacency but in the overarching control-flow topology. Malicious actors frequently reuse a core set of utility functions (e.g., for execution, encoding, or file access) but invoke them from different program locations, often inserting non-functional "junk" calls to thwart simple pattern matching.

Structural representations, by abstracting the trace into a graph or tree, capture these complex, non-local relationships. They model the program's architectural blueprint—the "who-calls-whom" relationships—which is a far more fundamental and stable indicator of shared malicious logic. This inherent robustness to superficial code reordering and obfuscation is the primary reason for their superior performance.

Table 3: The top 3 representation methods for each classifier, ranked by overall performance across all datasets.

Classifier	Top 3 Optimal Representation Methods
K-Means	Tree-GAT, Graph-GAT, Tree-Kernel
Mean Shift	Tree-GAT, CBOW, GloVe
Random Forest	Tree-GCN, Graph-GCN, Tree-GAT
SVM	Tree-GAT, Graph-GIN, Tree-GIN

7.2 Key Insight 2: Hierarchical Context is Crucial, Granting Trees an Edge

While both graph and tree models prove effective, our results show that tree-based representations yield top-tier overall performance (Table 3). This underscores the critical importance of **hierarchical context** in distinguishing WebShell families.

The reason for this advantage lies in what each structure preserves. A standard FCG is an aggregate view, merging all invocations of a function into a single node. It shows that function $A()$ called $B()$, but loses the context of how that call occurred. In contrast, a FCT is acyclic and preserves the precise execution path. Each node in an FCT represents a unique function invocation within a specific call stack.

This distinction is vital. A polymorphic function like $eval()$ might be used for different purposes depending on its caller. An FCT disambiguates these cases, representing $handler\_1() \rightarrow eval()$ and $handler\_2() \rightarrow eval()$ as distinct nodes with different parent-child relationships. This fine-grained, contextual fingerprint provides a more potent feature set for learning.

7.3 Key Insight 3: GNNs are the Premier Architecture for Learning Behavioral Topologies

Among all models, GNNs emerge as the most powerful and stable architecture, particularly the GAT and GCN. The theoretical underpinning is clear: GNNs are purpose-built to learn from relational data via a message-passing paradigm that explicitly models network topology. Unlike classic methods (e.g., Graph Kernels) that count predefined substructures, GNNs automatically learn the most discriminative structural motifs.

The particular strength of GAT stems from its attention mechanism. In a WebShell's call graph, not all function calls are equally important; calls to $system()$, $assert()$, or $base64\_decode()$ are far more salient than generic operations. GAT learns to assign higher attention weights to these diagnostically critical nodes and edges, effectively focusing on the parts of the call graph that best define a family's malicious signature.

7.4 Practical Implications and Guidance

Our findings offer a clear, actionable guide for building more intelligent malware defense systems.

Implications for Threat Discovery and Operational Use. As expected, supervised classifiers achieve higher overall

Models	Representation Methods		DS3											
			KS-ACC	KS-NMI	MS-ACC	MS-NMI	RM-ACC	RM-PRE	RM-REC	RM-FSC	SVM-ACC	SVM-PRE	SVM-REC	SVM-FSC
Sequence-Based	Word2Vec	concat	0.742	0.869	0.492	0.681	0.974	0.967	0.972	0.970	0.972	0.938	0.947	0.942
		avg	0.819	0.917	0.536	0.704	0.963	0.936	0.945	0.940	0.895	0.698	0.755	0.725
		concat&avg	0.709	0.844	0.517	0.721	0.959	0.935	0.949	0.942	0.655	0.291	0.385	0.331
	GloVe	concat	0.768	0.880	**0.625**	0.773	0.969	0.953	0.957	0.955	0.968	0.950	0.958	0.954
		avg	0.806	0.907	0.541	0.707	0.964	0.952	0.953	0.953	0.903	0.766	0.803	0.784
		concat&avg	0.727	0.857	0.611	0.768	0.961	0.924	0.942	0.933	0.710	0.324	0.416	0.365
	Bert	concat	0.719	0.852	0.534	0.691	0.973	0.969	0.973	0.971	0.962	0.946	0.951	0.949
		avg	0.665	0.817	0.534	0.694	0.942	0.902	0.906	0.904	0.938	0.882	0.909	0.895
		cls	0.640	0.812	0.570	0.705	0.945	0.910	0.921	0.915	0.929	0.830	0.857	0.843
	simCSE	concat	0.687	0.859	0.586	0.741	0.934	0.885	0.895	0.890	0.956	0.921	0.944	0.932
		avg	0.635	0.826	0.587	0.735	0.916	0.810	0.838	0.824	0.943	0.899	0.914	0.906
		cls	0.610	0.802	0.482	0.696	0.849	0.676	0.702	0.689	0.942	0.879	0.902	0.890
Graph-Based	Graph Kernel	Path	0.808	0.896	0.426	0.659	0.963	0.939	0.950	0.945	0.965	0.946	0.950	0.948
		Walk	0.818	0.905	0.500	0.671	0.960	0.929	0.936	0.932	0.965	0.942	0.948	0.945
		SubTree	0.876	0.933	0.397	0.638	0.959	0.929	0.940	0.935	0.936	0.850	0.885	0.867
	Graph Edit Distance		0.744	0.878	0.493	0.672	0.960	0.919	0.932	0.925	0.962	0.923	0.937	0.930
	GNN	GCN	0.786	0.911	0.621	**0.788**	**0.978**	0.973	**0.977**	**0.975**	0.969	0.957	0.968	0.962
		GAT	**0.926**	**0.960**	0.586	0.739	**0.978**	**0.973**	**0.975**	**0.974**	0.969	0.954	0.963	0.958
		GIN	0.628	0.857	0.465	0.692	0.971	0.961	0.967	0.964	**0.978**	**0.977**	**0.977**	**0.977**
	Graph Embedding (Graph2Vec)		0.792	0.897	0.619	0.769	0.961	0.945	0.950	0.947	0.957	0.918	0.936	0.927
Tree-Based	Tree Kernel	Path	0.809	0.897	0.392	0.643	0.956	0.922	0.944	0.933	0.965	0.951	0.952	0.952
		Walk	0.792	0.888	0.518	0.654	0.951	0.906	0.925	0.915	0.954	0.904	0.923	0.914
		SubTree	**0.895**	0.938	0.432	0.657	0.968	0.946	0.952	0.949	0.960	0.931	0.939	0.935
	Tree Edit Distance		0.741	0.878	0.499	0.697	0.964	0.941	0.954	0.947	0.967	0.941	0.954	0.948
	GNN	GCN	0.850	**0.939**	0.611	0.784	**0.979**	**0.977**	**0.979**	**0.978**	0.973	0.962	**0.969**	0.965
		GAT	**0.943**	**0.965**	**0.637**	**0.797**	**0.979**	**0.979**	**0.979**	**0.979**	**0.976**	**0.964**	0.968	**0.966**
		GIN	0.586	0.829	0.453	0.638	0.964	0.934	0.939	0.936	**0.976**	**0.973**	**0.975**	**0.974**
	Tree Embedding (Graph2Vec)		0.788	0.896	**0.638**	**0.786**	0.964	0.953	0.956	0.955	0.956	0.921	0.942	0.931

Figure 2: Performance comparison of representation methods on the *DS3 dataset*. Columns denote classifiers (KM: K-Means; MS: Mean-Shift; RF: Random Forest; SVM) and metrics (ACC: Accuracy; NMI: Normalized Mutual Information; PRE: Precision; REC: Recall; FSC: F1-Score).

Models	Representation Methods		Dataset 4											
			KS-ACC	KS-NMI	MS-ACC	MS-NMI	RM-ACC	RM-PRE	RM-REC	RM-FSC	SVM-ACC	SVM-PRE	SVM-REC	SVM-FSC
Sequence-Based	Word2Vec	concat	0.651	0.859	0.288	0.526	0.919	0.860	0.883	0.871	0.915	0.878	0.897	0.887
		avg	0.620	0.856	0.383	0.662	0.934	0.897	0.917	0.907	0.838	0.719	0.786	0.751
		concat&avg	0.496	0.791	0.396	0.673	0.942	0.901	0.917	0.908	0.418	0.158	0.224	0.185
	GloVe	concat	0.750	0.890	0.403	0.657	0.946	0.893	0.906	0.899	0.942	0.913	0.923	0.918
		avg	0.792	0.920	**0.445**	**0.717**	0.941	0.913	0.922	0.917	0.844	0.716	0.777	0.745
		concat&avg	0.648	0.860	0.421	0.700	0.943	0.906	0.917	0.912	0.489	0.185	0.267	0.219
	Bert	concat	0.594	0.817	0.416	0.688	0.933	0.891	0.903	0.897	0.893	0.866	0.883	0.875
		avg	0.603	0.822	0.368	0.669	0.853	0.782	0.808	0.795	0.821	0.730	0.784	0.756
		cls	0.579	0.813	0.388	0.669	0.861	0.808	0.825	0.816	0.822	0.735	0.791	0.762
	simCSE	concat	0.631	0.846	0.402	0.651	0.843	0.712	0.750	0.730	0.905	0.819	0.849	0.834
		avg	0.612	0.821	0.332	0.604	0.826	0.707	0.754	0.730	0.877	0.785	0.814	0.799
		cls	0.577	0.794	0.316	0.547	0.729	0.554	0.617	0.584	0.832	0.714	0.761	0.737
Graph-Based	Graph Kernel	Path	0.837	0.926	0.346	0.612	0.947	0.923	0.937	0.930	0.943	0.920	0.936	0.928
		Walk	0.768	0.896	0.375	0.633	0.946	0.918	0.937	0.927	0.946	0.914	0.930	0.922
		SubTree	0.829	0.925	**0.426**	**0.705**	0.925	0.897	0.914	0.905	0.895	0.834	0.868	0.851
	Graph Edit Distance		0.802	0.915	0.350	0.650	0.961	0.928	0.943	0.935	0.967	**0.948**	**0.957**	**0.953**
	GNN	GCN	0.821	0.933	0.413	0.703	0.963	**0.950**	**0.960**	**0.955**	0.949	0.923	0.933	0.928
		GAT	**0.872**	**0.945**	0.335	0.634	0.958	0.941	0.949	0.945	0.934	0.896	0.913	0.904
		GIN	0.536	0.831	0.309	0.591	0.944	0.925	0.937	0.931	0.953	0.936	0.952	0.944
	Graph Embedding (Graph2Vec)		0.515	0.847	0.329	0.686	**0.963**	**0.949**	**0.956**	0.953	**0.972**	**0.962**	**0.968**	**0.965**
Tree-Based	Tree Kernel	Path	0.728	0.876	0.360	0.597	0.942	0.890	0.914	0.902	0.940	0.880	0.903	0.891
		Walk	0.843	0.934	0.410	**0.705**	0.942	0.906	0.918	0.912	0.935	0.887	0.909	0.898
		SubTree	**0.895**	**0.938**	**0.432**	0.657	**0.968**	0.946	0.952	0.949	0.960	0.931	0.939	0.935
	Tree Edit Distance		0.848	0.929	0.330	0.652	0.948	0.923	0.934	0.928	0.947	0.932	0.946	0.939
	GNN	GCN	0.835	0.936	0.338	0.613	**0.963**	0.949	**0.967**	**0.958**	0.938	0.915	0.922	0.918
		GAT	**0.879**	**0.952**	0.371	0.674	0.959	**0.965**	0.949	**0.957**	**0.967**	0.933	0.944	0.939
		GIN	0.558	0.842	0.332	0.620	0.948	0.924	0.938	0.931	0.954	0.933	0.946	0.939
	Tree Embedding (Graph2Vec)		0.763	0.902	0.341	0.637	**0.963**	0.939	0.948	0.944	**0.969**	**0.953**	**0.964**	**0.959**

Figure 3: Performance comparison of all representation methods on the *DS4 dataset*.

Table 4: Optimal implementation strategies for sequence-based methods.

Method	Classifier	Optimal Strategy
CBOW/GloVe	KM/MS/RF	avg
	SVM	concat
BERT/SimCSE	All Classifiers	concat

Table 5: Optimal implementation strategies for graph- and tree-based methods.

Method	Classifier	Optimal Strategy
Graph Kernel	Unsupervised	Subtree Kernel
	Supervised	Path Kernel
Tree Kernel	All Classifiers	Subtree Kernel
GNNs	Unsupervised	GCN, GAT
	Random Forest	GAT
	SVM	GIN

performance than unsupervised clustering algorithms, highlighting the value of high-quality labels for building high-precision models. Thus, when a sufficient corpus of labeled data is available, supervised classification is the preferred approach. However, in real-world security operations, labels for emerging threats are often scarce or unavailable. This is where unsupervised methods become indispensable, as their ability to group samples by intrinsic behavioral similarity provides a direct pathway for discovering new or unknown WebShell families. Our results show that in this setting, the performance gap between structural and sequential representations is magnified, making the choice of a robust structural representation even more critical. Security teams can leverage this to automatically group new malware samples, flagging emergent clusters as potential zero-day threats requiring expert analysis.

Optimal Implementation Strategies. Achieving these results requires pairing the right abstraction with the right model variant. Our benchmark provides a clear roadmap (summarized in Tables 3, 4, and 5).

- **For overall performance**, a Tree-GAT model is the most consistent top performer across both supervised and unsupervised tasks.
- **For GNNs**, GAT and GCN are best for clustering, while GIN shows strength with SVMs in supervised settings.
- **For Graph Kernels**, Subtree Kernels are generally superior, especially for Tree Kernels where they are the optimal choice for all classifiers.
- **For sequence models**, the optimal aggregation strategy depends on the model architecture. For context-free embeddings like CBOW and GloVe, averaging the token embeddings of a trace is effective. For context-aware transformers like BERT and SimCSE, more sophisticated strategies like concatenating hidden states or using the final [CLS] token representation are superior.

8 Related Work

WebShell Detection Research on WebShell detection has predominantly focused on **binary classification**, distinguishing malicious from benign scripts. Early efforts relied on rule-based methods using signature matching, which proved ineffective against obfuscated or novel threats (Le et al. 2021; Hannousse and Yahiouche 2021). Subsequently, machine learning and deep learning techniques became mainstream, extracting lexical, statistical, or semantic features from source code or opcodes to train classifiers (Jinping et al. 2020; Pu et al. 2022; Shang et al. 2024; Zhang, Kang, and Wang 2025). Recently, Large Language Models (LLMs) have demonstrated strong zero-shot capabilities in this domain, achieving competitive performance without task-specific fine-tuning (Han et al. 2025).

However, while binary detection is well-studied, research on the more granular task of **WebShell family multi-classification** remains scarce. This gap is significant, as identifying the specific family of a WebShell is crucial for threat intelligence and targeted defense. A foundational contribution in this area is the MWF dataset (Zhao et al. 2024), which provided the first publicly available, family-annotated dataset of malicious WebShells, thereby enabling systematic research into multi-class classification, including ours.

Representation Learning for Program Behavior Our work is grounded in representation learning, which aims to transform complex, unstructured data like function call traces into meaningful vector embeddings. Inspired by natural language processing, early methods treat program traces as sentences. Classic techniques like Word2Vec (specifically, CBOW and Skip-gram) (Mikolov et al. 2013) and GloVe (Pennington, Socher, and Manning 2014) learn static, context-independent embeddings for each function. The advent of transformers led to powerful contextual models like BERT (Devlin et al. 2018), which capture deeper semantic relationships. More recently, contrastive learning methods such as SimCSE (Gao, Yao, and Chen 2021) have further improved the quality of sentence-level embeddings.

To capture the rich relational structure of function calls, we also explore graph-based methods. Traditional approaches include Graph Kernels, such as the Weisfeiler-Lehman (WL) kernel, which measure graph similarity by counting shared substructures (Shervashidze et al. 2011). Unsupervised methods like Graph2Vec learn embeddings for entire graphs by treating them as documents and their subgraphs as words (Narayanan et al. 2017). The current state-of-the-art, however, is dominated by Graph Neural Networks, which learn node and graph representations through iterative message passing. Our work benchmarks several prominent GNN architectures: Graph Convolutional Networks (Kipf and Welling 2016), Graph Attention Networks (Veličković et al. 2018), and Graph Isomorphism Networks (Xu et al. 2018).

9 Conclusion

In this paper, we establish the first systematic benchmark for fine-grained WebShell family classification, offering a robust empirical foundation for practical implementation.

Ethical Statement

This research is fundamentally aimed at generating a positive societal impact by enhancing cybersecurity against malicious WebShells, a class of malware that poses a direct threat to critical infrastructure, including government, financial, and healthcare systems. The primary benefit of this work is empowering security organizations to move beyond simple detection to a more nuanced, family-level understanding of threats. This capability translates directly into tangible societal goods: it enables faster incident response to minimize data breaches of sensitive records, aids law enforcement in attributing attacks, and helps preserve the integrity and public trust in essential digital services.

A core component of our ethical methodology was the deliberate decision to release only the dynamic function call traces, not the underlying source code. This approach provides the research community with a rich behavioral summary for analysis while intentionally withholding the full, executable malicious code. By doing so, we prevent the direct redistribution or weaponization of the original malware, ensuring that our dataset serves to strengthen defenses without creating new security risks.

We acknowledge the dual-use nature of cybersecurity research, where publicizing effective methods could inform adversarial strategies. However, we contend that the net effect of this open research is overwhelmingly positive for defenders. Our focus on dynamic behavior is inherently more robust against the common obfuscation techniques used by attackers. By providing a systematic framework and sharing our findings, we aim to level the playing field, giving defenders—especially those at smaller or under-resourced organizations—the tools and knowledge to adapt more quickly. We believe the societal benefits of advancing defensive capabilities through open, responsible research significantly outweigh the inherent risks.

References

Aboaoja, F. A.; Zainal, A.; Ghaleb, F. A.; Al-Rimy, B. A. S.; Eisa, T. A. E.; and Elnour, A. A. H. 2022. Malware detection issues, challenges, and future directions: A survey. *Applied Sciences*, 12(17): 8482.

Breiman, L. 2001. Random forests. *Machine learning*, 45(1): 5–32.

Comaniciu, D.; and Meer, P. 2002. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5): 603–619.

Cortes, C.; and Vapnik, V. 1995. Support-vector networks. *Machine learning*, 20(3): 273–297.

De Goër, F.; Rawat, S.; Andriesse, D.; Bos, H.; and Groz, R. 2018. Now you see me: Real-time dynamic function call detection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 618–628.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Feng, P.; Wei, D.; Li, Q.; Wang, Q.; Hu, Y.; Xi, N.; and Ma, J. 2024. GlareShell: Graph learning-based PHP webshell detection for web server of industrial internet. *Computer Networks*, 245: 110406.

Gao, T.; Yao, X.; and Chen, D. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, 6894–6910.

Han, F.; Zhang, J.; Deng, C.; Tang, J.; and Liu, Y. 2025. Can llms handle webshell detection? overcoming detection challenges with behavioral function-aware framework. *arXiv preprint arXiv:2504.13811*.

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

Jinping, L.; Zhi, T.; Jian, M.; Zhiling, G.; and Jiemin, Z. 2020. 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*, 251–259.

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

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

Ma, M.; Han, L.; and Zhou, C. 2024. Research and application of artificial intelligence based webshell detection model: A literature review. *arXiv preprint arXiv:2405.00066*.

MacQueen, J. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, 281–297. University of California Press.

Marzal, A.; and Vidal, E. 1993. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9): 926–932.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Narayanan, A.; Chandramohan, M.; Venkatesan, R.; Chen, L.; and Liu, Y. 2017. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*.

Pennington, J.; Socher, R.; and Manning, C. D. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 1532–1543.

Pu, A.; Feng, X.; Zhang, Y.; Wan, X.; Han, J.; and Huang, C. 2022. BERT-Embedding-Based JSP Webshell Detection on Bytecode Level Using XGBoost. *Security and Communication Networks*, 2022(1): 4315829.

Shang, M.; Han, X.; Zhao, C.; Cui, Z.; Du, D.; and Jiang, B. 2024. Multi-language webshell detection based on abstract

syntax tree and treelstm. In *2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 377–382. IEEE.

Shervashidze, N.; Schweitzer, P.; van Leeuwen, E. J.; Mehlhorn, K.; and Borgwardt, K. M. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9): 2539–2561.

Starov, O.; Dahse, J.; Ahmad, S. S.; Holz, T.; and Nikiforakis, N. 2016. No honor among thieves: A large-scale analysis of malicious web shells. In *Proceedings of the 25th International Conference on World Wide Web*, 1021–1032.

Tu, T. D.; Guang, C.; Xiaojun, G.; and Wubin, P. 2014. Webshell detection techniques in web applications. In *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 1–7. IEEE.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2018. Graph attention networks. *stat*, 1050(20): 10–48550.

Wrench, P. M.; and Irwin, B. V. 2015. Towards a PHP webshell taxonomy using deobfuscation-assisted similarity analysis. In *2015 Information Security for South Africa (ISSA)*, 1–8. IEEE.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*.

Xu, Y.; and Chen, Z. 2023. Family classification based on tree representations for malware. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, 65–71.

Zhang, Y.; Kang, H.; and Wang, Q. 2025. MMFDetect: Webshell Evasion Detect Method Based on Multimodal Feature Fusion. *Electronics*, 14(3): 416.

Zhao, Y.; Lv, S.; Long, W.; Fan, Y.; Yuan, J.; Jiang, H.; and Zhou, F. 2024. Malicious webshell family dataset for webshell multi-classification research. *Visual Informatics*, 8(1): 47–55.

Reproducibility Checklist

This paper:

Includes a conceptual outline and/or pseudocode description of AI methods introduced (Yes)

Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results (Yes)

Provides well marked pedagogical references for less-familiare readers to gain background necessary to replicate the paper (Yes)

Does this paper make theoretical contributions? (No)

If yes, please complete the list below.

All assumptions and restrictions are stated clearly and formally. (NA)

All novel claims are stated formally (e.g., in theorem statements). (NA)

Proofs of all novel claims are included. (NA)

Proof sketches or intuitions are given for complex and/or novel results. (NA)

Appropriate citations to theoretical tools used are given. (NA)

All theoretical claims are demonstrated empirically to hold. (NA)

All experimental code used to eliminate or disprove claims is included. (NA)

Does this paper rely on one or more datasets? (Yes)

If yes, please complete the list below.

A motivation is given for why the experiments are conducted on the selected datasets (Yes)

All novel datasets introduced in this paper are included in a data appendix. (Yes)

All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. (Yes)

All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations. (Yes)

All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available. (Yes)

All datasets that are not publicly available are described in detail, with explanation why publicly available alternatives are not scientifically satisficing. (NA)

Does this paper include computational experiments? (Yes)

If yes, please complete the list below.

This paper states the number and range of values tried per (hyper-) parameter during development of the paper, along with the criterion used for selecting the final parameter setting. (Yes)

Any code required for pre-processing data is included in the appendix. (Yes).

All source code required for conducting and analyzing the experiments is included in a code appendix. (Yes)

All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. (Yes)

All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from (Yes)

If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results. (NA)

This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks. (Yes)

This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics. (Yes)

This paper states the number of algorithm runs used to compute each reported result. (Yes)

Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information. (No)

The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank). (No)

This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments. (Yes)

A Implementation Details

This appendix provides detailed hyperparameter settings for our representation methods and downstream classifiers to ensure the reproducibility of our experiments. All models were implemented using Python 3.8 with PyTorch 1.12 and Scikit-learn 1.1. Experiments were conducted on a server equipped with an Intel Xeon Gold 6248R CPU, 256GB of RAM, and an NVIDIA A100 GPU.

A.1 Hyperparameters for Representation Methods

CBOW. We use the Word2Vec implementation from the Gensim library. The model is configured with a vector dimensionality of 128, a context window size of 5, 10 negative samples, and is trained for 100 epochs. A minimum word count of 2 was enforced. For the ‘concat‘ aggregation strategy, sequences longer than the maximum length of 256 are truncated.

GloVe. We use the official GloVe implementation. The model is configured with a context window size of 10 and an embedding dimensionality of 128. The weighting function parameter ‘xmax‘ is set to 100, and the exponent ‘alpha‘ is set to 0.75. The model was trained for 100 epochs using the Adam optimizer with a learning rate of 0.001 and a batch size of 512.

BERT & SimCSE. We utilize the ‘bert-base-uncased‘ architecture from the Hugging Face Transformers library as the foundation for both BERT and SimCSE. The model consists of 12 transformer layers, 12 attention heads, and a hidden size of 768, which is then projected to a final embedding of 128 dimensions via a linear layer. For pre-training, we construct a domain-specific corpus where each line contains two randomly selected function call sequences. The model is trained for 10 epochs with a batch size of 64, a learning rate of 2e-5, and the AdamW optimizer. For SimCSE, we use a dropout rate of 0.1 as the noise operator for the contrastive learning objective.

Graph Kernels. We use the ‘gklearn‘ library. For the Weisfeiler-Lehman (WL) subtree kernel, the number of iterations was set to 5. For the Random Walk (RW) kernel, the random walk length was set to 10.

Graph Edit Distance (GED). Our GED computation is implemented using the ‘gklearn‘ library with the following settings:

- **Edit Costs:** We define a constant edit cost vector of ‘[1, 1, 1, 1, 1, 1]‘ for node/edge deletion, insertion, and substitution. This uniform cost treats all structural changes equally.
- **GED Algorithm:** We use the `BIPARTITE` graph matching algorithm for its efficiency on large-scale graph data.

Graph2Vec. We use the official implementation of Graph2Vec. The model is configured with an embedding dimensionality of 128, 10 negative samples, and a WL subtree height of 3. It was trained for 100 epochs with a learning rate of 0.025.

Graph Neural Networks (GNNs). All GNN models (GCN, GAT, GIN) were implemented using PyTorch Geometric. Each model consists of 3 GNN layers followed by a global mean pooling layer and a 2-layer MLP head to produce the final 128-dimensional embedding. We trained for 200 epochs using the Adam optimizer with a learning rate of 0.001 and a batch size of 64. A dropout rate of 0.5 was applied after each GNN layer. For GAT, we used 4 attention heads.

A.2 Hyperparameters for Downstream Classifiers

To ensure a fair and robust comparison, we used a fixed set of optimized hyperparameters for our downstream classifiers, identified via grid search on a validation set.

K-Means. The number of clusters (‘k‘) is set to the ground-truth number of families in each dataset. We used the ”k-means++” initialization method and set ‘n_init‘ to 10 to ensure stability.

Mean-Shift. The bandwidth parameter was automatically estimated using the ‘estimate_bandwidth‘ function from Scikit-learn on a sample of the data.

Random Forest. The model is configured with 100 estimators (trees), a maximum depth of 10, and a minimum of 5 samples required to split an internal node.

Support Vector Machine (SVM). We use an SVM with a radial basis function (RBF) kernel. The regularization parameter ‘C‘ is set to 1.0, and the kernel coefficient ‘gamma‘ is set to ‘scale‘.

B Prompt Templates for LLM-Powered Data Augmentation

B.1 Prompt for Intra-Family Augmentation

The following prompt was used to generate new, diverse samples for existing WebShell families. The goal was to create traces that are behaviorally consistent with the target family while introducing syntactic variations.

Prompt for Intra-Family Data Augmentation

System Prompt: You are a cybersecurity expert specializing in malware analysis. Your task is to generate new, plausible dynamic function call traces for a specific WebShell family. The generated traces must be behaviorally consistent with the provided description and examples, but should introduce minor variations to enhance data diversity.
User Prompt: Based on the following behavioral profile and examples for the **[Family Name]** WebShell family, please generate 10 new and unique dynamic function call traces.
[Behavioral Description]
For Example: This family typically uses base64 decoding on POST data and then executes the result using an 'eval' or 'assert' call. It often includes file manipulation functions like 'fopen' and 'fwrite' for persistence.
[Example Traces]
For Example:
1. ["_main_", "base64_decode", "eval", "zend_fetch_r_post", ...]
2. ["_main_", "zend_compile_string", "assert", "base64_decode", ...]
Output:

B.2 Prompt for New Family Simulation

This prompt was designed to simulate adversarial innovation by instructing the LLM to create a novel WebShell family. This is achieved by blending the characteristics of two existing families, thereby generating data for zero-day threat scenarios.

Prompt for New Family & Zero-Day Simulation

System Prompt: You are an expert malware author. Your objective is to design a novel WebShell family by creatively blending the characteristics of two existing malware families. First, describe the core behavior and tactics of your new creation. Then, generate function call traces that reflect this new, hybrid behavior.
User Prompt: Design a new WebShell family that combines the stealth techniques of **[Family A Name]** with the command execution capabilities of **[Family B Name]**.

1. Provide a short description of the new family's behavior.
2. Generate 10 dynamic function call traces for this new family.

[Family A Profile: Name and Behavioral Description]
e.g., Family A (Stealthy Dropper): Focuses on obfuscation using string manipulation functions and avoids direct execution calls. It writes payloads to temporary files.
[Family B Profile: Name and Behavioral Description]
e.g., Family B (Powerful C2): Uses direct command execution via 'shell_exec' and 'system' and communicates over raw sockets.
[Example Traces from Family A & B]
Output:

C Annotation Examples

Table C.1: Examples of annotated samples with their complete identifiers and assigned family labels. A 'familyId' of -1 denotes an outlier.

Filemd5	Family ID
12b7340d1b8acf0fe2d78fce84bccf8c	1
1aba8701dcab6629caa9e21fc772b50e	2
28c5678442c6a3ee17290ece4d1c8904	3
00cd0f1bfda4903dba26541301c686ec	5
01625e53cb2d1275fbf4b2af0f6946e3	-1

D Dataset Details

Table D.1: Details of the four malicious WebShell family datasets used in our experiments. The complexity increases from DS1 to DS4.

Dataset	# Samples	Complexity	# Families	# Outliers
DS1	450	Low	21	1
DS2	550	Medium	37	10
DS3	1100	High	48	23
DS4	1600	High	81	28

E Results for DS1 and DS2

Models	Representation Methods		DS1											
			KS-ACC	KS-NMI	MS-ACC	MS-NMI	RM-ACC	RM-PRE	RM-REC	RM-FSC	SVM-ACC	SVM-PRE	SVM-REC	SVM-FSC
Sequence-Based	Word2Vec	concat	0.801	0.916	0.640	0.828	0.977	0.947	0.952	0.950	**0.980**	0.957	0.962	0.960
		avg	0.870	0.936	0.677	0.834	0.967	0.914	0.930	0.922	0.782	0.572	0.693	0.627
		concat&avg	0.810	0.895	0.640	0.809	0.971	0.946	0.958	0.952	0.416	0.200	0.325	0.247
	GloVe	concat	0.817	0.923	0.666	0.841	**0.980**	0.960	0.967	0.964	0.975	0.947	0.959	0.953
		avg	0.874	0.940	0.704	0.846	0.973	0.948	0.955	0.952	0.758	0.591	0.673	0.629
		concat&avg	0.846	0.908	0.677	0.835	0.977	0.952	0.954	0.953	0.322	0.168	0.275	0.209
	Bert	concat	0.782	0.905	0.598	0.807	0.979	0.959	0.963	0.961	0.977	0.948	0.960	0.954
		avg	0.882	0.936	0.674	0.828	**0.980**	0.966	**0.970**	**0.968**	0.973	0.938	0.950	0.944
		cls	0.859	0.930	**0.746**	**0.872**	0.977	0.951	0.955	0.953	0.979	0.959	0.970	0.964
	simCSE	concat	0.849	0.932	0.715	**0.880**	0.962	0.901	0.919	0.910	0.969	0.942	0.958	0.950
		avg	0.881	0.939	**0.746**	**0.877**	0.939	0.871	0.896	0.883	0.969	0.954	0.958	0.956
		cls	0.860	0.927	0.647	0.835	0.911	0.833	0.874	0.853	0.969	0.927	0.941	0.934
Graph-Based	Graph Kernel	Path	0.973	0.977	**0.738**	0.860	0.965	0.930	0.944	0.937	0.979	0.963	0.967	0.965
		Walk	0.825	0.925	0.662	0.817	0.962	0.916	0.935	0.925	0.971	0.933	0.937	0.935
		SubTree	0.916	0.959	0.674	0.854	0.971	0.930	0.940	0.935	0.926	0.886	0.918	0.902
	Graph Edit Distance		0.911	0.947	0.658	0.830	0.969	0.937	0.949	0.943	0.977	0.947	0.951	0.949
	GNN	GCN	**0.980**	**0.985**	0.621	0.798	**0.988**	**0.988**	**0.988**	**0.988**	0.979	0.950	0.953	0.951
		GAT	**0.980**	**0.985**	0.693	0.848	**0.980**	**0.968**	0.968	0.968	**0.984**	**0.978**	**0.983**	**0.981**
		GIN	0.787	0.917	0.602	0.804	0.965	0.919	0.934	0.926	**0.980**	0.972	**0.977**	0.975
	Graph Embedding (Graph2Vec)		0.846	0.933	0.681	0.854	0.975	0.946	0.952	0.949	0.960	0.910	0.928	0.919
Tree-Based	Tree Kernel	Path	0.973	0.977	0.711	0.859	0.965	0.914	0.929	0.922	0.977	0.943	0.957	0.950
		Walk	0.646	0.818	0.670	0.821	0.958	0.898	0.918	0.908	0.963	0.919	0.946	0.932
		SubTree	0.916	0.958	0.628	0.823	0.969	0.933	0.944	0.939	0.952	0.911	0.935	0.923
	Tree Edit Distance		0.841	0.921	0.647	0.831	0.962	0.910	0.930	0.920	0.975	0.946	0.956	0.951
	GNN	GCN	0.958	0.976	0.681	0.866	**0.984**	**0.978**	**0.983**	**0.981**	**0.980**	**0.976**	0.975	**0.975**
		GAT	**0.980**	**0.985**	0.681	0.851	0.977	0.959	0.962	0.961	**0.988**	**0.988**	**0.988**	**0.988**
		GIN	0.752	0.891	0.643	0.833	0.971	0.926	0.946	0.936	**0.980**	0.967	0.972	0.969
	Tree Embedding (Graph2Vec)		0.874	0.941	0.677	0.856	0.977	0.956	0.961	0.958	0.967	0.928	0.945	0.936

Figure E.1: Performance comparison of all representation methods on the DS1 dataset.

Models	Representation Methods		DS2											
			KS-ACC	KS-NMI	MS-ACC	MS-NMI	RM-ACC	RM-PRE	RM-REC	RM-FSC	SVM-ACC	SVM-PRE	SVM-REC	SVM-FSC
Sequence-Based	Word2Vec	concat	0.785	0.910	0.529	0.732	0.967	0.938	0.950	0.944	0.966	0.923	0.933	0.928
		avg	0.878	0.935	0.547	0.753	0.973	0.955	0.964	0.960	0.917	0.814	0.855	0.834
		concat&avg	0.715	0.865	0.487	0.722	0.971	0.925	0.940	0.932	0.506	0.232	0.310	0.265
	GloVe	concat	0.784	0.904	0.516	0.714	0.962	0.919	0.940	0.929	0.970	0.929	0.941	0.935
		avg	**0.914**	0.954	**0.603**	**0.803**	0.975	0.965	0.969	0.967	0.961	0.936	0.945	0.941
		concat&avg	0.709	0.866	0.487	0.722	0.966	0.916	0.930	0.923	0.499	0.183	0.301	0.227
	Bert	concat	0.772	0.891	0.439	0.669	0.972	0.939	0.951	0.945	0.973	0.951	0.962	0.956
		avg	0.823	0.911	0.543	0.760	0.955	0.934	0.950	0.942	0.967	0.932	0.950	0.941
		cls	0.839	0.917	**0.582**	**0.782**	0.959	0.933	0.946	0.939	0.965	0.945	0.953	0.949
	simCSE	concat	0.748	0.880	0.539	0.703	0.906	0.782	0.830	0.805	0.937	0.853	0.886	0.869
		avg	0.803	0.904	0.456	0.694	0.918	0.832	0.858	0.845	0.933	0.856	0.886	0.871
		cls	0.743	0.887	0.454	0.598	0.908	0.773	0.816	0.794	0.933	0.863	0.892	0.877
Graph-Based	Graph Kernel	Path	0.846	0.935	0.508	0.727	0.977	0.968	0.971	0.969	0.979	0.971	0.975	0.973
		Walk	0.667	0.843	0.466	0.663	0.972	0.937	0.944	0.941	0.973	0.953	0.965	0.959
		SubTree	0.884	0.944	0.491	0.711	0.975	0.949	0.955	0.952	0.971	0.956	0.963	0.960
	Graph Edit Distance		0.752	0.890	0.531	0.754	0.967	0.924	0.938	0.931	0.971	0.927	0.936	0.931
	GNN	GCN	0.903	0.956	0.543	0.756	0.980	0.974	0.977	0.975	0.972	0.935	0.943	0.939
		GAT	0.908	**0.966**	0.545	0.755	**0.983**	**0.979**	**0.982**	**0.980**	**0.980**	0.964	0.970	0.967
		GIN	0.624	0.832	0.500	0.707	0.968	0.949	0.956	0.953	**0.984**	**0.983**	**0.983**	**0.983**
	Graph Embedding (Graph2Vec)		0.785	0.894	0.524	0.749	0.962	0.916	0.928	0.922	0.971	0.942	0.954	0.948
Tree-Based	Tree Kernel	Path	0.848	0.936	0.516	0.706	0.973	0.946	0.956	0.951	0.977	**0.972**	0.975	0.973
		Walk	0.471	0.704	0.394	0.621	0.952	0.898	0.907	0.902	0.969	0.940	0.951	0.945
		SubTree	**0.917**	**0.959**	0.551	**0.774**	0.978	0.960	0.971	0.966	**0.980**	0.971	**0.978**	**0.974**
	Tree Edit Distance		0.734	0.882	0.518	0.759	0.960	0.911	0.929	0.920	0.960	0.890	0.911	0.900
	GNN	GCN	0.860	0.947	**0.562**	0.757	**0.983**	**0.982**	**0.983**	**0.983**	0.973	0.950	0.960	0.955
		GAT	**0.924**	**0.967**	**0.562**	0.758	**0.984**	**0.982**	**0.984**	**0.983**	**0.980**	0.966	0.968	0.967
		GIN	0.619	0.833	0.524	0.743	0.974	0.962	0.966	0.964	**0.985**	**0.985**	**0.985**	**0.985**
	Tree Embedding (Graph2Vec)		0.671	0.864	0.493	0.738	0.973	0.959	0.966	0.962	0.959	0.936	0.949	0.942

Figure E.2: Performance comparison of all representation methods on the DS2 dataset.