

Learning to Triage Taint Flows Reported by Dynamic Program Analysis in Node.js Packages

Ronghao Ni^{*1} Aidan Z.H. Yang^{*†2} Min-Chien Hsu¹ Nuno Sabino¹
 Limin Jia¹ Ruben Martins¹ Darion Cassel^{†2} Kevin Cheang^{†2}

¹Carnegie Mellon University

{ronghaon, minchieh, nsabino, liminjia, rubenm}@andrew.cmu.edu

²Amazon Web Services

aidayang@amazon.com, {darion.cassel, kevin.kmcheang}@gmail.com

Abstract—Program analysis tools often produce large volumes of candidate vulnerability reports that require costly manual review, creating a practical challenge: how can security analysts prioritize the reports most likely to be true vulnerabilities?

This paper investigates whether machine learning can be applied to prioritizing vulnerabilities reported by program analysis tools. We focus on Node.js packages and collect a benchmark of 1,883 Node.js packages, each containing one reported ACE or ACI vulnerability. We evaluate a variety of machine learning approaches, including classical models, graph neural networks (GNNs), large language models (LLMs), and hybrid models that combine GNN and LLMs, trained on data based on a dynamic program analysis tool’s output. The top LLM achieves $F_1=0.915$, while the best GNN and classical ML models reaching $F_1=0.904$. At a less than 7% false-negative rate, the leading model eliminates 66.9% of benign packages from manual review, taking around 60 ms per package. If the best model is tuned to operate at a precision level of 0.8 (i.e., allowing 20% false positives amongst all warnings), our approach can detect 99.2% of exploitable taint flows while missing only 0.8%, demonstrating strong potential for real-world vulnerability triage.

I. INTRODUCTION

Program analysis tools can identify potential security vulnerabilities in software, but often produce a large volume of unconfirmed vulnerability reports that require time-intensive manual examination by security analysts. Previous user studies show that developers generally prefer tools that report potential vulnerabilities with low false positives, that too many false positives is a significant pain point, and that many developers avoid tools that have too many false positives, leading to tool disuse [1], [2]. One way to reduce false positives is to automatically synthesize proof-of-concept (PoC) exploits that demonstrate that the reported vulnerabilities can indeed be exploited by an attacker (e.g., [3]–[11]). However, this is challenging: typically PoC exploits cannot be automatically synthesized for a large fraction of the reported vulnerabilities. Even when a PoC exploit exists, analysts still need to review whether the exploit matches the program’s threat model.

Our work aims to address this limitation of program analysis tools. We apply machine learning-based triage to prioritize the reports of existing tools by predicting which reported vulnerabilities are most likely to lead to a real-world exploit—we maintain high precision while significantly improving recall. Analysts can then focus their resources on reviewing reports with high likelihood of being true vulnerabilities.

This work focuses on the Arbitrary Command Injection (ACI) and Arbitrary Code Execution (ACE) vulnerabilities [12], [13] in Node.js JavaScript packages. The Node.js runtime is one of the most popular frameworks for web, desktop, and mobile developers. Studies have found that the Node.js ecosystem is full of packages containing vulnerabilities [14]–[17]. ACI and ACE are high-severity vulnerabilities that allow an attacker to execute code or commands on the system running the application. Previous research has introduced automated methods to identify potential ACI and ACE vulnerabilities in JavaScript programs [3], [4], [18]–[20]. Many of these approaches use dynamic taint analysis to detect vulnerabilities, which attempts to trace the flow of data from attacker-controlled inputs through the program to critical APIs. Dynamic analyses may report many potentially dangerous flows, but may not always indicate which ones can actually be exploited (i.e., true positives). On the other hand, static analysis techniques often generate even more false positives, further complicating the triaging process.

Researchers have applied machine learning to vulnerability detection [21]–[25], often modelling it as a classification task focusing on end-to-end detection where machine learning (ML) models are trained directly on unfiltered source code or abstract representations such as abstract syntax trees (ASTs) or control flow graphs (CFGs). These approaches typically operate on limited contexts, such as single functions or small code snippets, due to architectural constraints or dataset granularity. However, ACE and ACI vulnerability patterns can involve complex data flow dependencies which are not readily inferred from source code or textual explanations. In contrast, dynamic taint analysis tools can cover entire codebases and identify vulnerabilities across complex data flow patterns. For instance, vulnerable ACE and ACI flows

^{*} Equal contribution.

[†] This work is unrelated to the authors’ employment at, or affiliations with, Amazon Web Services.

Preprint. Under review.

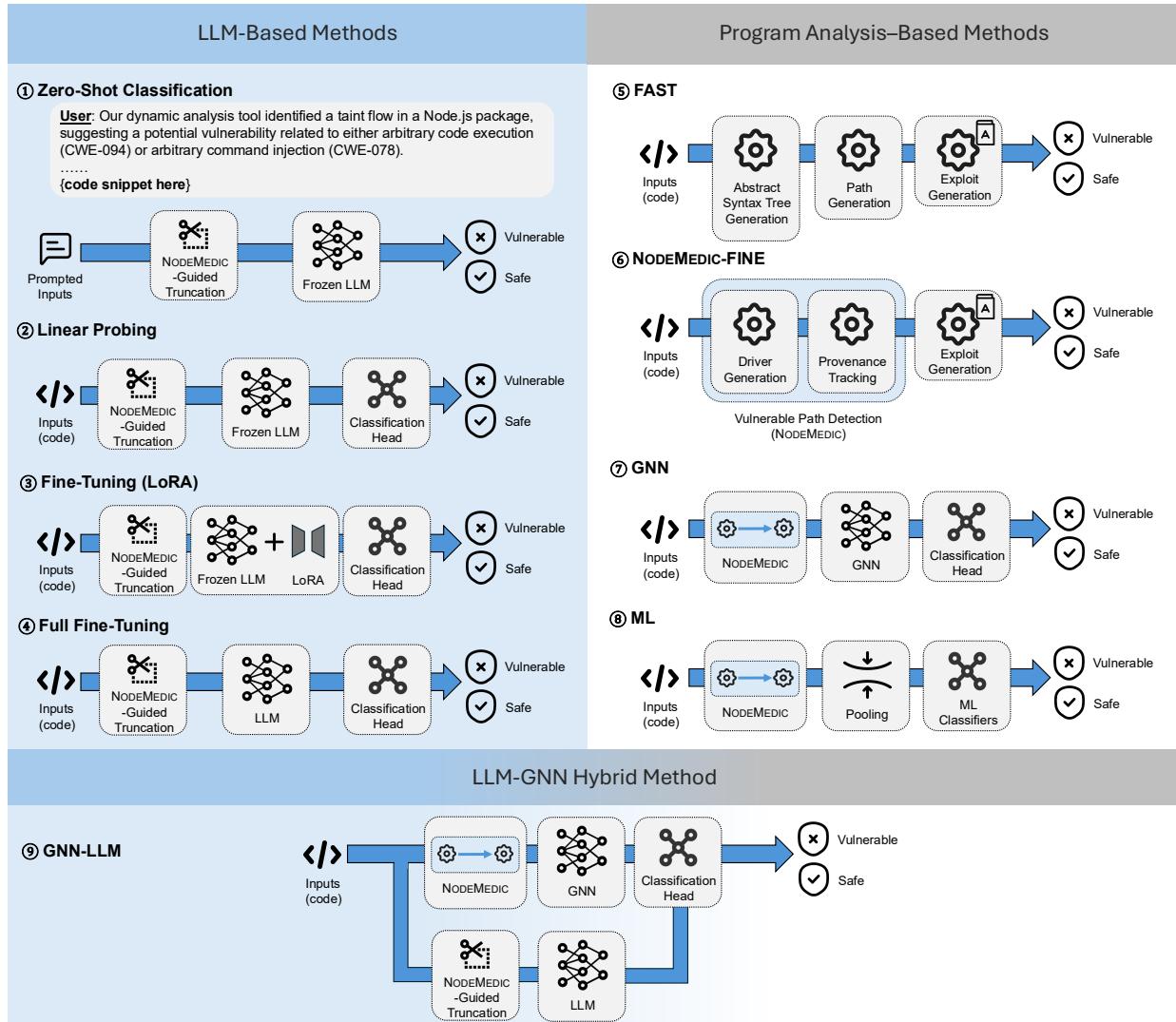


Fig. 1: All triage approaches evaluated

can be automatically discovered by dynamic taint analysis tools like NODEMEDIC [3] or the more recent NODEMEDIC-FINE [4], which output *taint provenance graphs* capturing all operations performed on attacker-controlled inputs flowing to the sink [3], [4].

This work investigates whether work done by dynamic taint analysis tools can be directly leveraged by ML models to triage ACE and ACI vulnerabilities in Node.js packages reported by these tools. Our insights are the following: **First**, traditional ML models (e.g., Graph Neural Networks—GNNs) can be configured to serve as a vulnerability report triage tool by performing binary classification on the provenance graphs. **Second**, since recent language models pre-trained on a large corpus of code tokens have a prior understanding of code structures and general vulnerability patterns, fine-tuning these models on a dataset annotated by a dynamic taint analysis tool to include specific vulnerability patterns results in strong vulnerability triage performance.

A summary of our evaluated machine learning approaches

for vulnerability triage is shown in Figure 1. LLM-only, which applies large language models directly to source code; ML, which uses classical ML models (Random Forest, XGBoost, Logistic Regression, and SVM) trained on features extracted from provenance graphs generated by the dynamic taint analysis tool NODEMEDIC-FINE; GNN, which uses graph neural networks on the same provenance graphs as in ML; GNN-LLM, a hybrid model that combines embeddings from both GNNs over provenance graphs and pre-trained LLMs over source code.

The baselines we compare against are program analysis and synthesis-based approaches for automatically synthesizing PoC exploits [3]–[5]. To evaluate our approaches we construct a benchmark, TRIAGE-JS, comprising 1,883 npm packages which NODEMEDIC-FINE successfully detected valid potential taint flows. Our best approach, a full-fine-tuned LLM with a classification head, achieves an average F1 score of 0.915. In comparison, NODEMEDIC-FINE without machine learning assistance achieves only $F_1 = 0.676$, failing to generate PoC

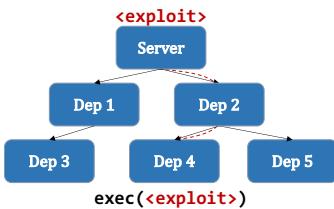


Fig. 2: Node.js attacker model

exploits for a large number of vulnerabilities. For practical vulnerability triage, at less than 7% false-negative rate *our best approach eliminates the need for manual review of 66.9% of non-vulnerable packages and takes around 60 ms per package.*

In summary, this paper makes the following contributions:

- **New benchmark for Node.js post-analysis triage.** We introduce TRIAGE-JS, a manually labeled benchmark of 1,883 Node.js packages, each containing one taint flow reported by a dynamic taint analysis tool. This dataset enables evaluation of ML-based triage methods for prioritizing taint flows reported by program analysis tools. This benchmark will be released after the responsible disclosure period is over for the novel identified vulnerabilities.
- **ML and hybrid taint flow triage.** We explore machine learning enhancements to program analysis tools for vulnerability report triage, showing that ML models can significantly reduce the effort spent on manually reviewing benign packages. In addition to standalone methods, we propose a hybrid technique (GNN-LLM) that combines GNNs trained on program analysis graphs with LLM code embeddings to provide a more comprehensive evaluation.
- **Empirical evaluation.** We perform a comprehensive evaluation on TRIAGE-JS. Our results show that all model families (classical ML, GNNs, and LLMs) can significantly reduce manual triage efforts compared to relying solely on program analysis tools. Among them, LLMs perform best, achieving the highest accuracy without requiring any analysis outputs.
- **Artifact availability.** To facilitate reproducibility, our tool and dataset will be made available.¹

II. BACKGROUND

We review Node.js’s threat model, program analysis tools for vulnerability detection and confirmation for Node.js packages, and existing ML approaches for vulnerability detection.

A. Node.js Package Threat Model

Node.js is built on top of the V8 JavaScript engine. Node.js developers combine code into *packages*, which can import other packages as *dependencies* to use their public APIs (exported functions). Node.js provides powerful sensitive APIs [26]–[29] that can dynamically generate and execute code and execute shell commands.

In a real-world attack, a Node.js package that unsafely uses sensitive Node.js APIs is included as a dependency of a victim

application as illustrated in Figure 2. An attacker can be any user communicating with the victim application. Attacker-controlled input is passed from the victim application to a sensitive API (e.g., `exec` [28]) via the dependency’s public API. We use the same model of the above scenario as prior work [3], [4], [30]: The attacker *directly* passes input to the dependency. We consider *all* public APIs of the dependency to be the attack surface of the package. This work focuses on two types of severe attacks: arbitrary code execution (ACE) [12] and arbitrary command injection (ACI) [13]. An attacker capable of these can launch other attacks, e.g., directory traversal [31], by extension.

application as illustrated in Figure 2. An attacker can be any user communicating with the victim application. Attacker-controlled input is passed from the victim application to a sensitive API (e.g., `exec` [28]) via the dependency’s public API. We use the same model of the above scenario as prior work [3], [4], [30]: The attacker *directly* passes input to the dependency. We consider *all* public APIs of the dependency to be the attack surface of the package. This work focuses on two types of severe attacks: arbitrary code execution (ACE) [12] and arbitrary command injection (ACI) [13]. An attacker capable of these can launch other attacks, e.g., directory traversal [31], by extension.

B. Dynamic Taint Analysis

Taint analysis specifies and checks policies governing sensitive dataflow with programs. A goal of the analysis is to detect flows from particular *sources*, e.g., API inputs, to *sinks*, e.g., sensitive APIs with dangerous capabilities, such as command execution. Taint analysis that runs during program execution is called dynamic taint analysis. Dynamic taint analysis has been particularly efficacious for detecting code vulnerabilities of JavaScript (c.f. [32]). Several tools perform taint analysis of Node.js packages [33]–[39]. We focus on leveraging NODEMEDIC-FINE [4], which implements a taint *provenance analysis*, storing a history of operations applied to tainted data as *provenance graph*, which can be used for further analysis.

C. Provenance Graph

Provenance graphs generated by NODEMEDIC-FINE capture the complete history of operations applied to tainted values during a program’s execution. Each node in a provenance graph represents a taint-related operation or value, while edges illustrate the flow of data between these nodes.

To illustrate, we reuse the toy example `toygrep` from prior work [3] to demonstrate how NODEMEDIC-FINE generates the provenance graphs using package source code and driver programs. This toy example (shown in Listing 1) exposes an API function called `grep`, which takes an argument `query` and executes the system command `grep [query]` without any sanitization. The driver program (shown in Listing 2) simulates the behavior of an external input by creating a variable `x` marked as tainted using the `__set_taint__` function. Then this tainted variable is passed as the `query` argument to the `grep` function, triggering the execution of the system command.

Figure 3 shows the provenance graph generated from the vulnerable toy package `grep` and its driver program. The graph illustrates how the input flows through the `grep` function call, is concatenated with the string `grep`, and ultimately reaches the sink, `exec`. Each node in a provenance graph captures key details about the operations and data flow within the program. Table I provides an overview of these attributes. Our work uses the provenance graph as input to a set of ML methods for vulnerability detection.

¹<https://doi.org/10.5281/zenodo.16758243>

```

1 function grep(query) {
2   exec("grep " + query);
3 }

```

Listing 1: Source code for the toy package `toygrep`

```

1 var PUT = require("toygrep");
2 var x = "tainted"; // {0:'0'}
3 __set_taint__(x);
4 try{PUT.grep(x);}
5 catch (e) {console.log(e)}

```

Listing 2: Driver program for the toy package `toygrep`

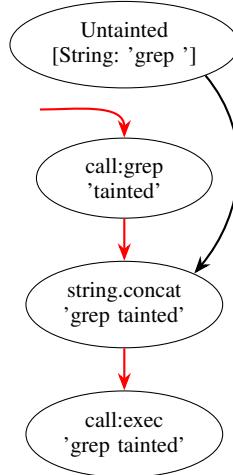


Fig. 3: Provenance graph generated from the `toygrep` package. Upper-left section contains source code (Listing 1); Bottom-left section contains the driver program (Listing 2); Right section presents the provenance graph. Ovals are node. Black edges are untainted, red edges are tainted flows.

TABLE I: Attributes of nodes in provenance graphs.

Attribute	Description
Operation	Type of operation, e.g., call or Untainted.
Value	Input provided to the operation.
File Path	File where the operation occurs.
Position	Start and end line/column in the file.
Tainted Status	Whether the data is tainted or untainted.
Flows From	Predecessor nodes in the data flow.
Sink Type	eval (ACE) or exec (ACI).

D. Vulnerability Confirmation Methods

NODEMEDIC-FINE [4] uses the provenance graph to synthesize a proof-of-concept exploit by mapping operations in the provenance graph to an SMTLIB constraint satisfaction formula, solving for package inputs that can deploy a test payload at the sink. The package is then executed with the synthesized inputs; successful payload execution (e.g., creating a target file) confirms the vulnerability.

Other tools generate candidate exploits without confirming them; FAST [5] uses constraint solving to generate *potential* exploits. Successful execution of the generated exploits could be used to confirm vulnerabilities, though FAST stops after exploit generation and thus can report false positives.

E. Machine Learning Vulnerability Detection

Classical approaches, such as Support Vector Machines (SVMs) and Random Forests, have been widely applied to program vulnerability detection [23], [40], [41]. These methods typically rely on a variety of features for classification, including program traces, call graphs, literals, variables, data types, operators, and statements. However, these methods

lack the ability to *simulate* program execution. More modern architectures, such as deep neural networks and graph neural networks, have been applied [24], [42], [43] to bridging this gap by fitting to highly nonlinear patterns and simulating dataflow, while large language models have shown promise in reasoning about program semantics [44], [45]. Prior work has explored a variety of usage modes: in zero-shot settings, LLMs are prompted to make predictions directly without task-specific tuning; in few-shot settings, they are provided with a handful of labeled examples to guide inference; and in finetuned settings, the model weights are updated on domain-specific data to specialize behavior.

Our work explores how these methods can be repurposed to assist triage rather than detection. Specifically, we use the taint provenance graphs generated by NODEMEDIC-FINE [4] and relevant source code as inputs to ML models, enabling them to predict which reported flows are truly exploitable and thus prioritize analyst attention more effectively.

III. DATASET AND METHODOLOGY

In this section, we detail our approaches for taint flow triage, as illustrated in Figure 1. We use NODEMEDIC-FINE as the underlying program analysis tool to identify candidate ACE and ACI vulnerabilities in Node.js packages, which are then triaged by the evaluated methods. We explain how we constructed our dedicated Node.js taint flow triage benchmark, TRIAGE-JS, on which all experiments are performed.

A. TRIAGE-JS: A Node.js Taint Flow Triage Benchmark

TRIAGE-JS contains 1,883 npm packages for which NODEMEDIC-FINE successfully detected potential ACE or ACI vulnerabilities and generated provenance graphs for the taint flows. We started from an initial set of 33,011 npm packages, obtained from the authors of NODEMEDIC-FINE. These packages were collected from the npm package registry, pre-filtered to include only those with at least one weekly download and at least one call to a NODEMEDIC-FINE-supported sink function. We also obtained a result of running NODEMEDIC-FINE’s vulnerability detection pipeline on this initial set from the authors of NODEMEDIC-FINE, which contained reported taint flows (potential ACE or ACI vulnerabilities) for 2,051 of these packages. However, 168 packages were excluded from the dataset due to issues such as the absence of sinks in the output provenance graph or invalid file paths or line numbers in the generated flow reports, leaving us with 1,883 packages.

Of the 1,883 potentially vulnerable npm packages, 664 were confirmed by NODEMEDIC-FINE’s exploit synthesis component as containing exploitable vulnerabilities. The remaining 1,219 packages were manually examined to determine exploitability by reviewers experienced in software security and program analysis. These reviewers inspected the taint flow, relevant code context, and potential for attacker-controlled input to reach security-sensitive sinks. Exploitability was assessed conservatively to minimize false positives.

Since ML methods require parts of the data for training and validation, we randomly divided the 1,883 package dataset into three subsets (train, validation, and test) in an 8:1:1 ratio. The training set includes 1,506 packages, the validation set includes 188 packages, and the testing set includes 189 packages. For models requiring training, the training set is used to train the model, while the validation set is used to select the best model before evaluation. All models are then evaluated on the same testing set to ensure consistent performance reporting. Table II provides a detailed overview of these dataset splits.

TABLE II: Overview of the dataset splits used in the evaluation. The table displays the total number of packages in each split (*train*, *validate*, and *test*), along with the number of vulnerable packages in each split, categorized into ACE and ACI. Numbers in parentheses indicate the count of vulnerable packages within each split.

Split	Total (Vuln)	ACE (Vuln)	ACI (Vuln)
train	1,506 (989)	255 (176)	1,251 (813)
validate	188 (124)	36 (23)	152 (101)
test	189 (137)	38 (29)	151 (108)
total	1,883 (1,250)	329 (228)	1,554 (1,022)

B. LLM-based Methods

Common practices in using large language models (LLMs) for classification tasks typically fall into two categories:

- 1) **Zero-shot and few-shot prompting**, where the model is provided with code snippets along with carefully designed natural language prompts to identify security risks. This approach benefits from LLMs’ generalization ability but often struggles with nuanced vulnerabilities that require deeper program understanding.
- 2) **Fine-tuning**, where the model is trained on labeled datasets to learn domain-specific patterns. Fine-tuning can significantly improve detection accuracy but comes with high computational costs and data collection challenges. Beyond full fine-tuning, there are also lightweight fine-tuning methods, such as LoRA fine-tuning [46], [47] and fine-tuning on only selected layers. Commonly, a language model can either be fine-tuned on a text generation objective or used as a classifier by attaching a classification head on top of the pre-trained model. We adopt the latter approach, treating the models as classifiers whose outputs are logits used for prediction.

To systematically evaluate LLMs in taint flow triage, we assess several models under different settings, including zero-shot classification, linear probing, LoRA fine-tuning [46], [47], and full fine-tuning. We exclude few-shot learning in this work because the large size of potentially vulnerable package code snippets makes it difficult to fit multiple samples into a reasonable context window. The code snippet given to LLMs is the file that contains potential sinks. If the file is too long to fit within the predefined context length, we truncate it to 1,024 tokens, taking code immediately around the sink. This is

based on our observations that the majority of vulnerable logic is local to the sink and the surrounding code in the dataset.

1) *Zero-Shot Classification* (① in Figure 1): In this setting, we use an auto-regressive generation head that enables the LLM to generate a textual response indicating whether a given JavaScript package contains vulnerabilities. Zero-shot classification relies entirely on the LLM’s pre-trained knowledge and ability to generate a relevant answer token by token in an auto-regressive manner. For this method, we prompt the LLM with the following query:

User: Our dynamic analysis tool identified a taint flow in a Node.js package, suggesting a potential vulnerability related to either arbitrary code execution (CWE-094) or arbitrary command injection (CWE-078). While the tool attempts to confirm vulnerabilities by generating exploits, this approach may miss some cases. I hope you can assist with triaging and classification by predicting whether the vulnerability is exploitable.

I have extracted relevant parts of the code from the file containing the sink, along with surrounding lines for context. After reasoning about the snippet, please output “Yes” if you believe it contains an exploitable vulnerability, or “No” if you believe it is not exploitable.

{code snippet here}

For models that run locally, we disable sampling during generation to ensure deterministic results. For models that require a cloud-based API, the classification responses may vary slightly across multiple runs. Even when a model is instructed to output only “Yes” or “No”, models trained with a Chain-of-Thought (CoT) objective [48] (often referred to as reasoning models) may still produce additional text, enclosed in special tokens, as part of their internal reasoning process. We disregard these additional tokens and only consider the final answer for evaluation. The outputs are filtered based on the presence of “Yes” which is considered vulnerable, while all other cases, including those that generate neither “Yes” nor “No” are considered non-vulnerable.

2) *Linear Probing* (② in Figure 1): Instead of using a generation head, we attach a classification head on top of the base LLMs and fine-tune the model for the taint flow triage task while keeping the base LLMs frozen (not involved in training). In our setup, the classification head is just a single linear layer (an affine linear transformation). The classification head takes the embedding from the last non-padding position of the output from the last attention layer as input and produces an output shape of 2 logits, representing the two classes (vulnerable or non-vulnerable). We use the cross-entropy loss function to train the classification head with weights that correspond to the class imbalance in the dataset.

3) *LoRA Fine-Tuning* (③ in Figure 1): LoRA (Low-Rank Adaptation) fine-tuning [46], [47] offers a lightweight approach to adapting LLMs without updating all parameters. Instead of modifying the entire model, LoRA injects low-rank adapters into selected layers of the model. These adapters are small, low-rank matrices that are learned during fine-tuning. Similar to the previous method, we attach a classification head on top of the base LLMs and fine-tune the model for the taint flow triage task. However, in this case, we only update the low-rank adapters and the classification head, while the base LLMs remain frozen. The same cross-entropy loss function is used to train the classification head.

4) *Full Fine-Tuning* (④ in Figure 1): In full fine-tuning, we update all parameters of the LLM and the classification head using our labeled benchmark TRIAGE-JS, applying the same cross-entropy loss function as previously described.

During the fine-tuning of all the aforementioned LLM-based methods, all frozen parameters are stored in 4-bit NormalFloat (NF4) precision for memory efficiency [47], while the trainable parameters are in 16-bit BrainFloat (BF16) precision.

C. GNN and Classical ML Methods

GNN and ML utilize the provenance graphs (as described in Section II-C) created by NODEMEDIC-FINE’s taint provenance tracking component in both the training and inference pipelines. The operation, tainted status of the arguments, and sink type of each node are used as inputs. Additionally, the vulnerability type is included as an input for the entire graph. The 100 most common operations in our dataset (as described in Section 3.1) are assigned class numbers 0 to 99. Class 100 is designated for less frequent operations, while class 101 is used for empty or missing operation attributes in the provenance graphs. Tainted statuses are encoded as class 0 for `False` (untainted), class 1 for `True` (tainted) and class 2 for missing attributes. Sink types are represented with class 0 for `spawn`, class 1 for `exec`, class 2 for `Function`, and class 3 for `eval`. Vulnerability types are encoded as class 0 for ACE and class 1 for ACI vulnerabilities.

Each attribute is represented as a one-hot vector, where the corresponding class has a value of 1, and all other classes have a value of 0. The four one-hot vectors are then concatenated to form the embedding for a single node in the graph. Together, the graph’s node connections and the embeddings of its nodes make up the complete representation of the graph.

1) *GNN* (⑦ in Figure 1): The GNN component in the GNN method starts with a Gated Graph Sequence Neural Network (GGNN) [49], which is a specialized type of neural network designed to learn from graph-structured data by capturing dependencies and relationships between nodes. The GGNN works by iteratively passing messages along edges, enabling each node to gather information from its neighbors and update its representation based on the graph’s structure and features. In the final step, the learned abstract node embeddings are combined into a graph-level representation using Global Attention Pooling [49], resulting in the final graph embedding.

The graph embedding is then fed into a classification head to predict vulnerability.

2) *ML* (⑧ in Figure 1): Classical ML methods only take the node embeddings of the graph into account, and discard the edges. The embeddings of all nodes are first fed into a pooling layer to create a unified shape embedding vector that represents the entire graph, regardless of the number of nodes. The pooled embedding is then passed through machine learning classifiers to predict the vulnerability.

D. GNN-LLM Hybrid Methods

We evaluate a hybrid approach (⑨ in Figure 1) that combines NODEMEDIC-FINE’s vulnerability path detection, graph neural networks (GNNs), and large language models (LLMs). This method, referred to as GNN-LLM, aims to integrate the strengths of both program analysis and LLMs to enhance taint flow triage. This hybrid approach constructs a joint representation by concatenating the embedding produced by the GNN with the embedding produced by the LLM. The combined vector is then passed to a classification head, which is trained to predict whether the input contains a vulnerability. Prior work [24] has shown that fusing heterogeneous features in this way can enhance downstream performance by leveraging diverse representation spaces. The full model is trained end-to-end, with the GNN and LLM components updated simultaneously.

IV. EXPERIMENTAL SETUP

We will first outline the model selection and implementation (Section IV-A) used in our evaluation. Next, we will discuss the evaluation metrics (Section IV-B) and the system configuration (Section IV-C).

A. Model Selection and Implementation

We directly report results of NODEMEDIC-FINE’s exploit synthesis obtained from NODEMEDIC-FINE’s authors. We downloaded FAST [50] and it ran against TRIAGE-JS with the `-X` flag enabled to turn on exploit generation. Additionally, for FAST, we used the `-t` flag to specify vulnerability types as `os_command` and `code_exec`, which correspond to ACI and ACE vulnerabilities, respectively.

For classical ML methods, we evaluate logistic regression, support vector machine (SVM), random forest, and XGBoost. These methods are trained on provenance graphs generated by NODEMEDIC-FINE. For logistic regression, SVM, and random forest experiments, we used classes from the `scikit-learn` library, while the `xgboost` package was used for XGBoost experiments. For these machine learning baseline models, default hyperparameters were applied. The GNN method is implemented using the `torch-geometric` library [51]. The model is trained with a learning rate of 0.001, a batch size of 64, and a weight decay rate of 0.1. The GNN model is trained for 150 epochs, with early stopping based on validation F1 scores.

We experiment with several language models and their different pretrained versions: DeepSeek-R1-Distill-Qwen-14B,

TABLE III: Comparison of LLM Models Across Different Experiment Settings.

LLM Model	Local / Cloud	# Parameters	Reasoning	Experiments Conducted			
				Zero-Shot	Linear Probing	LoRA FT	Full FT
OpenAI o4-mini-high	Cloud	Unknown	Yes	✓	✗	✗	✗
DeepSeek-R1-0528	Cloud	671B	Yes	✓	✗	✗	✗
Gemini 2.5 Pro	Cloud	Unknown	Yes	✓	✗	✗	✗
OpenAI GPT-4.1	Cloud	Unknown	No	✓	✗	✗	✗
DeepSeek-R1-Distill-Qwen-14B	Local	14B	Yes	✓	✓	✓	✗
DeepSeek-R1-Distill-Llama-8B	Local	8B	Yes	✓	✓	✓	✓
DeepSeek-R1-Distill-Qwen-7B	Local	7B	Yes	✓	✓	✓	✓
Llama-3.1-8B-Instruct	Local	8B	No	✓	✓	✓	✓
Qwen2.5-Coder-14B-Instruct	Local	14B	No	✓	✓	✓	✗
Qwen2.5-Coder-7B-Instruct	Local	7B	No	✓	✓	✓	✓

DeepSeek-R1-Distill-Llama-8B, DeepSeek-R1-Distill-Qwen-7B [52], Llama-3.1-8B [53], Qwen2.5-Coder-14B, and Qwen2.5-Coder-7B [54]. We use the publicly available implementations and parameters from Hugging Face. For Qwen and Llama models, we use the instruct-tuned versions for zero-shot evaluation, and the corresponding base models for finetuning with a classification head. Llama-3.1-8B-Instruct, Qwen2.5-14B-Instruct, and Qwen2.5-7B-Instruct are among the top four downloaded text generation models on Hugging Face [55] as of August 2025. We exclude the top-ranked model, GPT-2, as it is too small in size for our evaluation. For Qwen2.5-14B-Instruct and Qwen2.5-7B-Instruct, we use the coder variants, which are specifically designed for code-related tasks. We also include the DeepSeek-R1-Distill versions of these models, which are distilled from the much larger DeepSeek-R1 model.

We also evaluate zero-shot performance using several models accessed via cloud APIs: OpenAI o4-mini-high, DeepSeek R1-0528 [52], Gemini 2.5 Pro [56], and OpenAI GPT-4.1 [57]. These models are accessed through OpenRouter [58]. We select them to cover a range of commercial LLMs that are widely used in practice and represent different model families and deployment tiers, from lightweight variants like o4-mini-high to frontier models such as GPT-4.1 and Gemini 2.5 Pro.

For LoRA finetuning, we are using a rank of 128 and an alpha of 64 for all experiments. A study on LoRA’s hyperparameters is presented in Section V-C. All LLM methods that require fine-tuning are trained with a batch size of 2 per device, a learning rate of 1e-5, and a weight decay rate of 0.01. The training is conducted for 3 epochs, with early stopping based on validation F1 scores. An overview of the LLM models and their experiment settings is provided in Table III.

B. Evaluation Metrics

NODEMEDIC-FINE, FAST, and zero-shot LLM models are evaluated directly on the test dataset, while other models that require training are trained on the training dataset and validated on the validation dataset during the training process. The best model from training is then evaluated on the test dataset. To assess the performance of each method, we follow prior work on vulnerability detection [21], [24], [25], [59], and employ the following metrics: $F1 = \frac{TP}{TP+0.5\cdot(FP+FN)}$,

Precision = $\frac{TP}{TP+FP}$, Recall = $\frac{TP}{TP+FN}$, and Accuracy = $\frac{TP+TN}{TP+TN+FP+FN}$ (where TN is true negative, TP is true positive, FP is false positive, and FN is false negative). For all methods, we report the average of the metrics across five runs, each with a different random seed (2025 through 2029). The average and variance of the metrics are reported in Section V. For precision-recall curves shown in Section V-E, we also compute average precision (AP) across five runs at each recall level. The average precision is defined as $AP = \sum_{n=1}^N (R_n - R_{n-1}) \cdot P_n$, where P_n , R_n is the respective precision and recall at threshold index n .

C. System Configuration

Evaluations that require only CPUs are conducted on a computing cluster. Each task runs individually in a virtually isolated environment with a 2-core CPU, which is part of an AMD EPYC 7742 processor, and 16 GB of RAM. The timeout for each task is set to 36 hours. For experiments that require GPUs, except for those where LLMs are undergoing full finetuning, each task is executed in a virtual environment with one NVIDIA H100 (80GB) GPU, two Intel Xeon 8480C PCIe Gen5 CPUs (each with 56 cores running at 2.0/3.8 GHz), and 2 TB of RAM. For LLMs that require full fine-tuning, we use a computing cluster with two NVIDIA H100 (80GB) GPUs, two Intel Xeon 8480C PCIe Gen5 CPUs (each with 56 cores running at 2.0/3.8 GHz), and 2 TB of RAM.

V. EVALUATION

Our experiments address the following research questions:

RQ1: Method Effectiveness. How effective are machine learning methods, including classical models, GNNs, and LLMs, at assisting in triaging taint flows reported by taint analysis tools?

RQ2: Comparing Graph- and LLM-Based Models. How do the predictions of graph-based models (e.g., GNN and classical ML methods) and LLM-based methods differ, and does combining GNN with LLMs improve triage?

RQ3: Comparison of LLM Usage Strategies. How do different large language models and usage strategies (e.g., zero-shot inference vs. fine-tuning) perform in the context of vulnerability triage?

TABLE IV: F1 Score (F1), Precision (Prec), Recall (Rec), and Accuracy (Acc) for all methods. Higher metrics indicate better performance. For models with multiple configurations or usage variations, we report the results for the setup that achieves the highest F1 score. All metrics are reported as the mean and variance over five runs with different random seeds. A dash (–) indicates that the corresponding metric is undefined due to division by zero or the model has only one setup. **Bold** indicates the best F1 score within each group; underline indicates the best F1 score overall. F1 confidence intervals are visualized using horizontal bars centered at the mean, computed as $\bar{x} \pm 2.776 \cdot \text{SD}$ (95% confidence with $n = 5$, Student’s t distribution). Intervals outside the fixed axis range (0.85–0.95) are clipped for display.

Model	Best Setup	F1			Prec		Rec		Acc	
		Mean	Var	CI Bar	Mean	Var	Mean	Var	Mean	Var
Random ($P_{vuln} = 1/2$)	—	0.592	0.00e+00	——	0.725	0.00e+00	0.500	0.00e+00	0.500	0.00e+00
Random ($P_{vuln} = 989/1506$)	—	0.689	0.00e+00	——	0.725	0.00e+00	0.657	0.00e+00	0.571	0.00e+00
Random ($P_{vuln} = 1$)	—	0.841	0.00e+00	——	0.725	0.00e+00	1.000	0.00e+00	0.725	0.00e+00
Random ($P_{vuln} = 0$)	—	0.000	0.00e+00	——	—	—	0.000	0.00e+00	0.275	0.00e+00
NODEMEDIC-FINE	—	0.676	0.00e+00	——	1.000	0.00e+00	0.511	0.00e+00	0.646	0.00e+00
FAST	—	0.647	0.00e+00	——	0.957	0.00e+00	0.489	0.00e+00	0.614	0.00e+00
GNN	—	0.886	1.71e-04	●—●	0.914	6.99e-05	0.858	3.36e-04	0.839	3.02e-04
Random Forest	Avg Pooling	0.900	8.37e-06	●—●	0.917	1.99e-07	0.883	2.66e-05	0.857	1.40e-05
XGBoost	Avg Pooling	0.904	0.00e+00	●—●	0.917	0.00e+00	0.891	0.00e+00	0.862	0.00e+00
Logistic Regression	Max Pooling	0.892	0.00e+00	●—●	0.909	0.00e+00	0.876	0.00e+00	0.847	0.00e+00
SVM	Max Pooling	0.898	0.00e+00	●—●	0.898	0.00e+00	0.898	0.00e+00	0.852	0.00e+00
OpenAI o4-mini-high	Zero-shot	0.805	7.49e-05	——	0.889	5.64e-05	0.736	2.77e-04	0.742	7.53e-05
DeepSeek R1-0528	Zero-shot	0.857	5.70e-05	●—●	0.824	1.04e-04	0.893	3.36e-04	0.784	1.03e-04
Gemini 2.5 Pro	Zero-shot	0.851	1.85e-04	●—●	0.769	3.95e-04	0.953	1.49e-04	0.758	6.08e-04
OpenAI GPT-4.1	Zero-shot	0.858	9.20e-06	●—●	0.778	7.45e-06	0.956	2.66e-05	0.770	2.25e-05
DeepSeek-R1-Distill-Qwen-14B	LoRA FT	0.866	1.29e-04	●—●	0.861	2.90e-04	0.872	2.29e-04	0.804	2.94e-04
DeepSeek-R1-Distill-Llama-8B	LoRA FT	0.889	2.70e-04	●—●	0.878	1.93e-04	0.901	5.75e-04	0.837	5.37e-04
DeepSeek-R1-Distill-Qwen-7B	Full FT	0.915	1.72e-04	●—●	0.900	1.90e-04	0.931	2.56e-04	0.875	3.72e-04
Llama-3.1-8B-(Instruct)	LoRA FT	0.909	9.46e-05	●—●	0.888	6.67e-05	0.930	2.29e-04	0.865	1.90e-04
Qwen2.5-Coder-14B-(Instruct)	LoRA FT	0.863	1.18e-04	●—●	0.846	8.87e-05	0.880	4.95e-04	0.797	2.04e-04
Qwen2.5-Coder-7B-(Instruct)	Full+GNN FT	0.902	6.26e-05	●—●	0.882	4.27e-04	0.923	3.89e-04	0.854	1.62e-04

RQ4: Method Efficiency. How do machine learning methods compare to traditional taint analysis approaches in terms of training overhead and latency?

RQ5: Triage Precision–Recall Trade-offs. What are optimal trade-offs between reviewing a high percentage of non-vulnerable packages and having a high chance of missing vulnerabilities?

A. RQ1: Methods Effectiveness

To understand the effectiveness of classical ML, GNN, and LLM-based methods for triaging taint flows, we compare their classification performance using F1 score, Precision, Recall, and Accuracy, defined in Section IV-B. Higher values indicate better ability to distinguish exploitable taint flows from false alarms. Table IV summarizes the results across all evaluated methods. For methods with multiple configurations (e.g., LLMs), only the best-performing setup is shown here; full results are included in Section V-B.

As baselines, we include four naive random predictors seeded with different probabilities of predicting a package to be vulnerable. Random ($P_{vuln} = 1/2$), Random ($P_{vuln} = 989/1506$), Random ($P_{vuln} = 1$), and Random ($P_{vuln} = 0$) represent uncalibrated guessing, empirical prior matching, and all-positive or all-negative predictors, respectively.

1) *Program Analysis-Based Approaches:* Traditional program analysis methods, such as FAST and NODEMEDIC-FINE’s PoC exploit synthesis, have high precision, but low recall, indicating that they often miss vulnerabilities due to failed PoC exploit synthesis. NODEMEDIC-FINE has better precision and recall than FAST, because FAST only generates potential exploits, some of which do not work.

Classical ML methods and GNN models significantly augment traditional program analysis tools. The GNN achieves an F1 score of 0.886, while classical ML classifiers reach F1 scores in the 0.892–0.904 range. This indicates that applying machine learning methods to the outputs of dynamic taint analysis leads to high triage accuracy and recall without significantly sacrificing precision.

2) *LLM-based Methods:* LLM-based methods exhibit significant variability depending on whether they are used in a zero-shot setting or fine-tuned with different strategies. As shown in Table IV, zero-shot models often yield substantially lower F1 scores despite their large model sizes, highlighting a persistent gap between non-fine-tuned and fine-tuned approaches. A more detailed analysis is provided in Section V-C. Additionally, incorporating GNN embeddings into LLMs does not consistently lead to performance improvements, as discussed in Section V-B. Due to resource constraints, we only evaluated the larger models with LoRA-based fine-tuning,

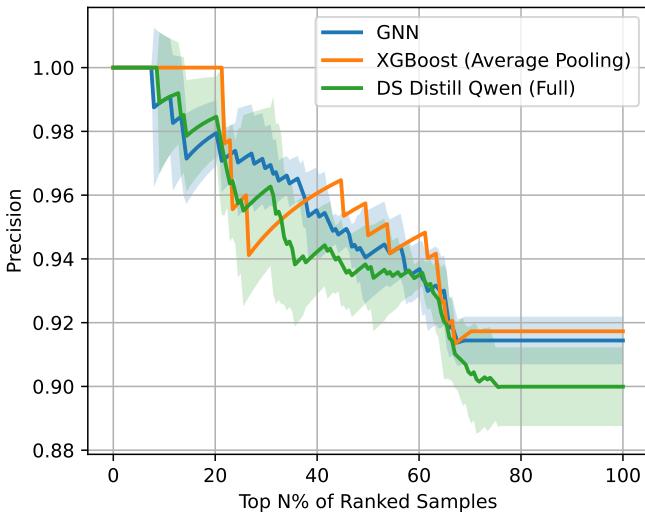


Fig. 4: Precision of the top N% outputs for GNN, XGBoost (average pooling) and full fine-tuning of DeepSeek-R1-Distill-Qwen-7B. The shaded area represents the standard deviation across five random seeds.

which may limit their performance. Among fine-tuned models, smaller variants with full fine tuning (e.g., Qwen-2.5-Coder-7B and Llama-3.1-8B) outperform larger ones with LoRA-based fine-tuning (e.g., Qwen-2.5-Coder-14B). Overall, DeepSeek-R1-Distill-Qwen-7B achieves the best average F1 score, while Llama-3.1-8B(-Instruct) and Qwen2.5-Coder-7B(-Instruct) exhibit mostly overlapping confidence intervals with it, suggesting comparable performance.

Overall, LLM-based methods (e.g., DeepSeek-R1-Distill-Qwen-7B with Full Fine-Tuning, F1: 0.915) outperform all traditional program analysis approaches, classical ML models, and GNN-based methods. This performance gain is likely attributable to (1) the fact that vulnerabilities often exhibit local patterns that LLMs can model well, and (2) the rich prior knowledge encoded through large-scale pretraining. Among classical ML methods, XGBoost with average pooling achieves an F1 score of 0.904. Although slightly lower than the best-performing LLM, its score falls within the 95% confidence interval of the top LLM, making it statistically comparable.

3) *Prioritizing Exploitable Flows in Ranked Outputs:* In addition to standard classification metrics, we assess triage effectiveness by computing the precision at the top N% of model outputs, where N% refers to the highest-scoring predictions based on model confidence. This is motivated by prior work showing that developers typically focus on only the top 5 findings when addressing issues [60], making this evaluation more reflective of real-world usage. In Figure 4, we include the GNN, XGBoost, and DeepSeek-Distill-Qwen-7B models, which are the best-performing models in their respective families. Precision remains high at lower values of N%, indicating that exploitable flows tend to be ranked near the top. This finding demonstrates the practical value of

TABLE V: Performance comparison between fully fine-tuned LLMs with and without GNN components. Each cell shows the average and variance on separate lines.

Model	F1	Precision	Recall	Accuracy
DeepSeek-R1-Distill-Llama-8B				
Full	0.6837 ($\pm 1.47\text{e-}01$)	0.6627 ($\pm 1.41\text{e-}01$)	0.7153 ($\pm 1.68\text{e-}01$)	0.6794 ($\pm 5.39\text{e-}02$)
Full + GNN	0.5512 ($\pm 1.48\text{e-}01$)	0.6920 ($\pm 1.53\text{e-}01$)	0.5226 ($\pm 1.68\text{e-}01$)	0.5820 ($\pm 5.44\text{e-}02$)
DeepSeek-R1-Distill-Qwen-7B				
Full	0.9153 ($\pm 1.72\text{e-}04$)	0.8999 ($\pm 1.90\text{e-}04$)	0.9314 ($\pm 2.56\text{e-}04$)	0.8751 ($\pm 3.72\text{e-}04$)
Full + GNN	0.9116 ($\pm 1.25\text{e-}04$)	0.8901 ($\pm 3.74\text{e-}05$)	0.9343 ($\pm 3.46\text{e-}04$)	0.8688 ($\pm 2.44\text{e-}04$)
Llama-3.1-8B				
Full	0.5046 ($\pm 1.61\text{e-}01$)	0.6672 ($\pm 1.53\text{e-}01$)	0.5299 ($\pm 2.32\text{e-}01$)	0.5386 ($\pm 4.76\text{e-}02$)
Full + GNN	0.8312 ($\pm 5.58\text{e-}04$)	0.8492 ($\pm 3.62\text{e-}04$)	0.8161 ($\pm 2.91\text{e-}03$)	0.7608 ($\pm 6.49\text{e-}04$)
Qwen2.5-Coder-7B				
Full	0.8962 ($\pm 6.17\text{e-}04$)	0.8931 ($\pm 5.91\text{e-}05$)	0.9007 ($\pm 2.68\text{e-}03$)	0.8497 ($\pm 9.88\text{e-}04$)
Full + GNN	0.9016 ($\pm 6.26\text{e-}05$)	0.8821 ($\pm 4.27\text{e-}04$)	0.9226 ($\pm 3.89\text{e-}04$)	0.8540 ($\pm 1.62\text{e-}04$)

our evaluated models, which produce confidence scores and therefore enable more effective prioritization during triage.

RQ1 Summary: Fine-tuned LLM-based methods outperform all traditional program analysis tools, classical ML models, and GNNs in detecting exploitable taint flows. The best-performing model, DeepSeek-R1-Distill-Qwen-7B (Full FT), achieves the highest F1 score. Classical ML models such as XGBoost also perform competitively, outperforming GNN method. These findings highlight the effectiveness of applying language models to program analysis outputs for vulnerability triage.

B. RQ2: Comparison of Graph- and LLM-Based Models

To understand whether Graph- and LLM-based models complement each other, we compare the predictions of fully fine-tuned LLMs with and without GNN embeddings and summarize the results in Table V. We evaluate the performance of each model on the test split, reporting F1 score, Precision, Recall, and Accuracy. Each cell shows the average and variance over five runs with different random seeds. We observe that Llama exhibits high instability during training, with F1 score variances around 0.1. Focusing on the Qwen models, we find that adding GNN to the LLM does not always improve performance. Next, we investigate why.

1) *Prediction Agreement Analysis:* To quantitatively compare the predictions of GNNs and LLMs, we analyze the test split predictions from both models with Cohen’s Kappa coefficient [61], which measures the agreement between two raters. A Kappa value of 1 indicates perfect agreement, while a value of 0 indicates no agreement beyond chance. We also include the predictions of the classical ML model XGBoost

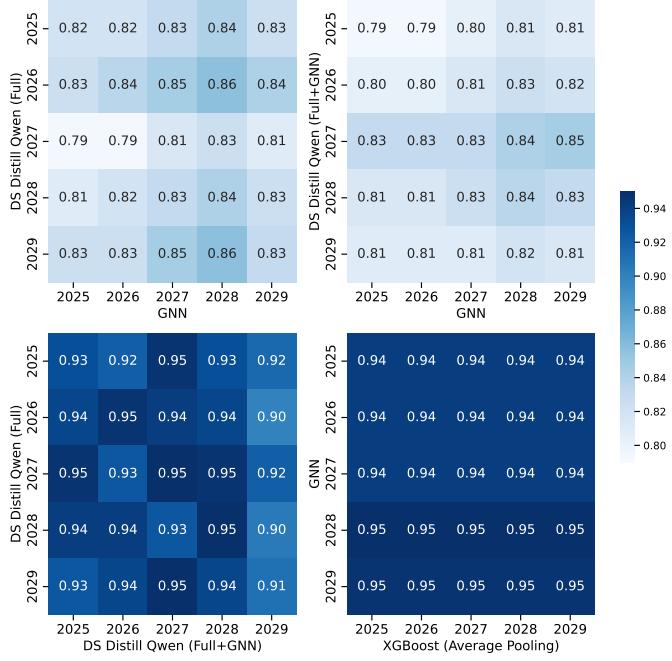


Fig. 5: Cohen’s Kappa coefficient for the predictions of GNN, DeepSeek-R1-Distill-Qwen-7B, and XGBoost on the test split. The Kappa values are computed across all combinations of random seeds for each method. The higher the Kappa value, the more agreement there is between the two methods.

with average pooling, which is the best-performing classical ML model in our experiments as a representative baseline for comparison against LLM- and GNN-based models, since both XGBoost and GNN operate on taint flows reported by NODEMEDIC-FINE. For LLM methods, we select the best-performing model, DeepSeek-R1-Distill-Qwen-7B (DS Distill Qwen) for comparison. We calculate the Kappa coefficient for the following four pairs of methods:

- DS Distill Qwen (Full) vs. GNNs
- DS Distill Qwen (Full+GNN) vs. GNNs
- DS Distill Qwen (Full) vs. DS Distill Qwen (Full+GNN)
- GNNs vs. XGBoost (Average Pooling)

Figure 5 shows the Cohen’s kappa values for each pair of methods. The values range from 0.0 to 1.0, with higher values indicating greater agreement between the two methods. When comparing the LLM method DeepSeek-R1-Distill-Qwen-7B (Full) with GNNs, we observe an average kappa value of 0.83, suggesting strong agreement between the two. This indicates that both methods tend to make very similar predictions, which helps explain why combining their embeddings does not lead to significant performance improvement.

2) *Explaining Large Language Models via SHAP*: We then try to understand how LLMs make predictions based on the input code. For this evaluation, we use DeepSeek-R1-Distill-Qwen-7B as a representative model. We compare the Shapley value-based explanations generated by SHAP [62] (SHapley Additive exPlanations) of the classification-head-only fine-

```
'use strict';

var fs = require('fs');
var phantomPath = require('phantomjs-prebuilt').path || '/usr/local/bin/phantomjs';
var Promise = require('promise');
var script = fs.realpathSync(__dirname + '/detect-phantom.js');
var spawn = require('child_process').spawn;

exports.check = function check (uri, whitelist) {
    return new Promise(function (resolve, reject) {
        var args = [script, uri];
        if (whitelist) args.push(whitelist);

        var phantomjs = spawn(phantomPath, args);
        var buffer = '';

        phantomjs.on('data', function(data) { buffer += data; });

        phantomjs.on('exit', function(code){
            var stdout = buffer.split("\n");
            stdout.pop();
            resolve([code, stdout]);
        });
    });
}
```

Fig. 6: Shapley values of the full-fine-tuned DeepSeek-R1-Distill-Qwen-7B on an package *third-party-resources-checker*. The correct prediction is: “not vulnerable.” Red values indicate a positive contribution to the vulnerable prediction, while blue values indicate a negative contribution. Darker colors represent a higher absolute Shapley value.

tuning version with the full fine-tuning to see how the model’s focus shifts during training. SHAP is a method for interpreting machine learning models by quantifying the contribution of each feature to the model’s predictions. Based on cooperative game theory, it assigns Shapley values to features, which measure each feature’s contribution to a collective outcome, offering insights into feature importance.

Specifically, we choose the packages where, in the testing split, the classification-head-only fine-tuning makes incorrect predictions, while the fully fine-tuned version makes correct predictions. The classification head-only fine-tuning does not alter the LLM’s core components, preserving most of its pre-trained parameter values. We found that for non-vulnerable packages, the sink function *spawn* (ACI) consistently exhibited a high Shapley value, strongly contributing to the model’s classification as “not vulnerable.” Figure 6 is an example of a non-vulnerable package *third-party-resources-checker*, where the presence of *spawn* is the most significant feature in the prediction after full fine-tuning. This suggests that the presence of *spawn* is associated with safer execution patterns compared to other process creation functions. One key reason for this is how *spawn* handles its arguments. Unlike *exec*, which interprets a string directly as shellcode, *spawn* takes a command and its arguments separately as elements of an array, and takes a second configuration object that only allows for shell metacharacter evaluation if explicitly configured. This design significantly reduces the risk of ACI attacks because an attacker must control both the command array and the configuration object (unless the package itself takes the unsafe action to allow shellcode evaluation). The need to control multiple arguments was specifically noted by NodeMedic-

FINE as a challenge for confirming exploitability of *spawn*.

For other cases, no specific pattern stands out, and most tokens contribute only marginally to the model’s predictions. This suggests that, apart from certain key indicators like *spawn*, the taint flow triage relies on a distributed set of features rather than any single dominant token. The Shapley values for these other tokens tend to be small and dispersed, indicating that their individual influence on the final prediction is limited. This behavior supports the idea that security vulnerabilities often result from complex interactions among different parts of the code, rather than being linked to the presence or absence of a single token.

RQ2 Summary: Combining GNN embeddings with fully fine-tuned LLMs provides limited additional benefit. The high average Cohen’s kappa value of 0.83 indicates strong agreement between their predictions, suggesting that the two models rely on overlapping signals.

C. RQ3: Comparison of LLM Usage Strategies.

Our experiments show that the performance of LLMs in the taint flow triage task is very sensitive to both the base model and the selected usage strategy (e.g., zero-shot, various fine-tuning methods). Table VI presents average F1 scores and variances across five random seeds for all LLM-based approaches under different usage configurations, including zero-shot and various fine-tuning strategies.

TABLE VI: F1 scores (with variances) for different large language models (LLMs) under various usage strategies: zero-shot inference, linear probing (LP), LoRA-based fine-tuning (LoRA FT), and full model fine-tuning (Full FT). A dash (–) indicates that the method is not evaluated for the model.

Zero-shot	LP	LoRA FT	Full FT
OpenAI o4-mini-high			
0.805 ($\pm 7.5\text{e-}05$)	–	–	–
DeepSeek R1-0528			
0.857 ($\pm 5.7\text{e-}05$)	–	–	–
Gemini 2.5 Pro			
0.851 ($\pm 1.9\text{e-}04$)	–	–	–
OpenAI GPT-4.1			
0.858 ($\pm 9.2\text{e-}06$)	–	–	–
DeepSeek-R1-Distill-Qwen-14B			
0.816 ($\pm 7.4\text{e-}05$)	0.788 ($\pm 3.2\text{e-}04$)	0.866 ($\pm 1.3\text{e-}04$)	–
DeepSeek-R1-Distill-Llama-8B			
0.793 ($\pm 8.1\text{e-}05$)	0.773 ($\pm 8.7\text{e-}04$)	0.889 ($\pm 2.7\text{e-}04$)	0.684 ($\pm 1.5\text{e-}01$)
DeepSeek-R1-Distill-Qwen-7B			
0.723 ($\pm 8.4\text{e-}04$)	0.779 ($\pm 5.4\text{e-}04$)	0.845 ($\pm 2.8\text{e-}04$)	0.915 ($\pm 1.7\text{e-}04$)
Llama-3.1-8B(-Instruct)			
0.303 ($\pm 1.0\text{e-}03$)	0.794 ($\pm 3.6\text{e-}04$)	0.909 ($\pm 9.5\text{e-}05$)	0.505 ($\pm 1.6\text{e-}01$)
Qwen2.5-Coder-14B(-Instruct)			
0.679 ($\pm 2.7\text{e-}04$)	0.729 ($\pm 1.6\text{e-}03$)	0.863 ($\pm 1.2\text{e-}04$)	–
Qwen2.5-Coder-7B(-Instruct)			
0.526 ($\pm 4.2\text{e-}04$)	0.744 ($\pm 5.3\text{e-}04$)	0.859 ($\pm 3.6\text{e-}04$)	0.896 ($\pm 6.2\text{e-}04$)

1) *Zero-Shot*: In a zero-shot setting, DeepSeek R1-0528, Gemini 2.5 Pro, and OpenAI GPT-4.1 achieve the highest performance, all with F1 scores exceeding 0.85, demonstrating strong out-of-the-box reasoning capabilities for vulnerability

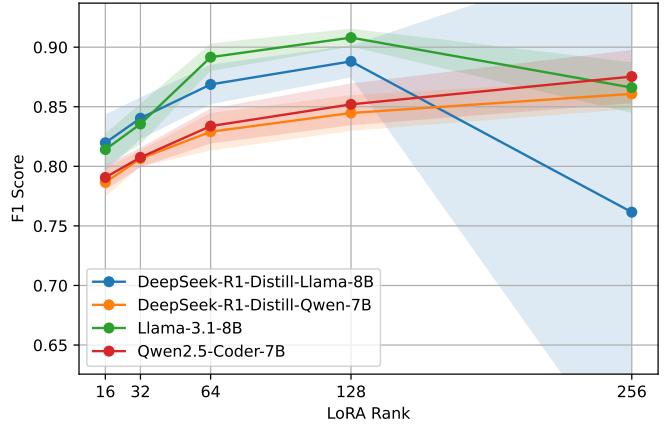


Fig. 7: Comparison of F1 scores across different LoRA ranks for four language models: DeepSeek-R1-Distill-Llama-8B, DeepSeek-R1-Distill-Qwen-7B, Llama-3.1-8B, and Qwen2.5-Coder-7B. Each point shows the mean F1 score on the test split over five random seeds, with shaded regions indicating standard deviation.

triage. Among large commercial models, OpenAI o4-mini-high, as a smaller variant of OpenAI’s o-series models, though pre-trained for longer thinking before responding, performs the worst in zero-shot settings, with an F1 score of 0.805.

Comparing these commercial models with smaller open-source models, we find that all except OpenAI’s o4-mini-high outperform the open-source models. The open-source models achieve F1 scores ranging from 0.526 (Qwen2.5-Coder-7B-Instruct) to 0.816 (DeepSeek-R1-Distill-Qwen-14B). Among models with the same or similar architectures, the DeepSeek-R1-Distill variants significantly outperform their non-distilled counterparts, suggesting that the distillation process effectively improves performance on the taint flow triage task.

2) *LoRA versus Full Fine-Tuning*: It is widely believed that a properly hyper-parameter-tuned Low-Rank Adaptation (LoRA) method can achieve performance similar to or even better than full fine-tuning [46], [47], [63] in the text generation setting. As shown in Figure 7, we observed similar trends in our evaluation of Llama-family models: LoRA fine-tuning outperforms full fine-tuning in terms of average F1 score for both DeepSeek-R1-Distill-Llama-8B and Llama-3.1-8B. Full fine-tuning on Llama-family models introduces significant instability, with variances reaching 0.1, indicating high sensitivity to random seeds. In contrast, LoRA fine-tuning yields much lower variance at smaller LoRA ranks. However, this stability benefit diminishes at larger ranks. For example, DeepSeek-R1-Distill-Llama-8B again shows unstable behavior at a LoRA rank of 256.

Llama and Qwen models respond differently to increases in LoRA rank. For Llama-family models (DeepSeek-R1-Distill-Llama-8B and Llama-3.1-8B), performance peaks at a LoRA rank of 256, with further increases offering no benefit. In contrast, Qwen-family models (DeepSeek-R1-Distill-Qwen-

7B and Qwen2.5-Coder-7B) show continued improvement in F1 score as LoRA rank increases, suggesting that these models benefit more from higher-rank adapters.

RQ3 Summary: LLM performance in vulnerability triage is highly sensitive to both model choice and usage strategy. In zero-shot settings, most large commercial models performs better than smaller open-source models. DeepSeek-R1-Distill models outperform their non-distilled counterparts, suggesting that distillation effectively enhances performance. LoRA fine-tuning can sometimes achieve similar or even better performance than full fine-tuning for Llama-family models. Qwen-family models benefit from higher LoRA ranks, while still underperforming full fine-tuning.

TABLE VII: Training and inference times for different models and methods. The inference time is measured per sample in a batched setting: calculated by dividing batch inference time by batch size for methods that support batch inference. Model saving and loading times are not included in the time measurement. *Other ML classifiers* refer to Random Forest, XGBoost, Logistic Regression, and SVM.

Model	Computation Time	
	Training	Inference
NODEMEDIC-FINE	0min	0.79s
FAST	0min	31.6s
GNN	25min	0.79s
Other ML Classifiers	22min	0.79s
DeepSeek-R1-Distill-Qwen-14B		
-zero-shot	0min	15.37s
DeepSeek-R1-Distill-Llama-8B		
-zero-shot	0min	22.72s
DeepSeek-R1-Distill-Qwen-7B		
-zero-shot	0min	5.04s
Llama-3.1-8B(-Instruct)		
-zero-shot	0min	0.48s
-linear-probing	11mins	0.11s
-lora-ft	34mins	0.11s
-full-ft	70mins	0.10s
Qwen2.5-Coder-14B(-Instruct)		
-zero-shot	0min	0.50s
-linear-probing	19mins	0.21s
-lora-ft	58mins	0.21s
Qwen2.5-Coder-7B(-Instruct)		
-zero-shot	0min	0.25s
-linear-probing	10mins	0.10s
-lora-ft	30mins	0.10s
-full-ft	60mins	0.06s

D. RQ4: Methods Efficiency

To compare the computational efficiency of LLM-based methods against traditional and ML-enhanced program analysis approaches, we analyze both training time (pre-prediction) and inference time (prediction latency), as presented in Table VII. All full fine-tuning of the LLMs is performed on two (2) GPUs, unlike other methods for LLMs where experiments are conducted using one (1) GPU. For FAST, the inference

time is the median overhead reported in the paper [5] that introduces FAST. The graph generation times of NODEMEDIC-FINE represent the average tool runtime reported in the paper [3], which is 0.79 seconds per output. Some methods of DeepSeek-R1-Distill models (e.g., Full FT) are not reported because they share the same model structure as the non-distilled models; their performance is expected to be similar.

Full fine-tuning or LoRA fine-tuning of LLMs usually takes more training time than ML-enhanced program analysis tools. However, the inference time for fine-tuned LLMs is quicker than that of both program analysis methods and ML-enhanced program analysis methods when processing in batches. FAST and NODEMEDIC-FINE provide a PoC exploit, while the other methods cannot.

RQ4 Summary: LLM-based methods have slightly higher training costs compared to ML-enhanced program analysis methods, but remain within a practical range. When it comes to inference, fine-tuned LLMs are up to 10 times faster than both program analysis methods and ML-enhanced program analysis methods.

E. RQ5: Triage Precision–Recall Trade-offs

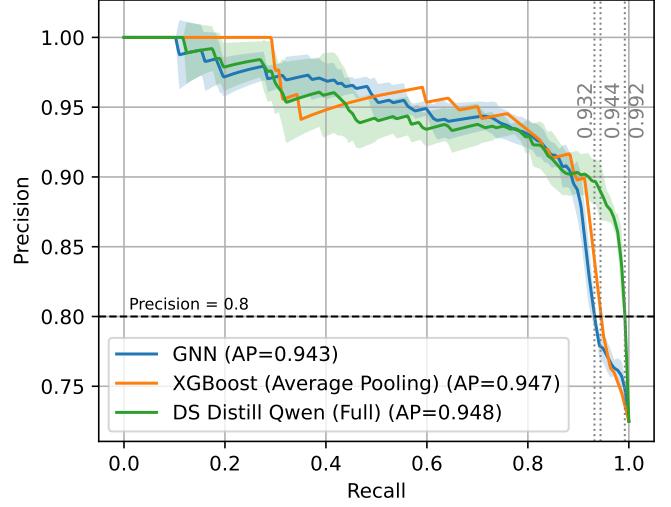


Fig. 8: Precision-recall curves for GNN, XGBoost (average pooling) and full fine-tuning of DeepSeek-R1-Distill-Qwen-7B. AP stands for Average Precision. The shaded area represents the standard deviation across five random seeds.

Vulnerability detection and triage tools try to balance between requiring review of a high percentage of non-vulnerable packages (false positives) and having a high chance of missing vulnerabilities (false negatives). We compare such tradeoffs of our methods and identify the optimal operating point. We compute a precision-recall (PR) curve using the standard scikit-learn implementation. Since different seeds may yield curves with varying recall points, we interpolate all PR curves onto a fixed recall axis using linear interpolation. This alignment

enables us to compute the mean and standard deviation of precision values across seeds at each recall level.

Figure 8 shows the precision-recall curves for the GNN, XGBoost (average pooling), and DeepSeek-R1-Distill-Qwen-7B (Full FT) models, which are the best-performing models in their respective families. The curves of all three models behave similarly in the lower-recall region. However, in the high-recall region (where recall > 0.9), the DeepSeek-R1-Distill-Qwen-7B (Full FT) model achieves substantially higher recall at each precision level. Based on prior user studies [1], developers generally prefer tools with precision above 0.8.² At the point on the precision-recall curve where precision reaches 0.8 (based on linear interpolation), the LLM model achieves 0.992 recall, while the GNN and XGBoost models reach only 0.932 and 0.944, respectively. In real-world terms, this suggests that if the model is configured to operate at a threshold corresponding to 0.8 precision, limiting false positives to 20% of alerts, it can successfully identify 99.2% of true vulnerabilities, significantly reducing the risk of missing critical issues.

RQ5 Summary: Our precision-recall curve analysis shows that when the best LLM model is configured to operate at a precision of 0.8, an accuracy level preferred by developers, it can detect 99.2% of all vulnerabilities while missing only 0.8% of exploitable taint flows.

F. Threats to Validity

The evaluation across all configurations of our technique relies on the dataset of 1,883 npm packages, which contain potential vulnerabilities identified by NODEMEDIC-FINE. When we obtained the dataset, the labeling of true positives was incomplete; 1,239 of these packages were not labeled. Two authors of the paper manually labeled these cases with cross-checking. Each manual confirmation is accompanied by an actual exploit. However, some vulnerabilities could still be missed due to human error.

The LLMs' performance could be impacted if their training data included vulnerability reports on packages in the TRIAGE-JS dataset. Although the source code of our 1,883 npm packages may have been released prior to our evaluated LLM's training cut-off dates (with the oldest being January 2023), the majority of the vulnerabilities in them were unlikely to be available prior to those dates. In particular, the vulnerability reports of the 606 vulnerable packages that we manually confirm and all but 35 from NODEMEDIC-FINE's set of auto-confirmed packages have not been released at the time of submission.

VI. RELATED WORK

A. Taint analysis of Node.js packages

Several tools perform dynamic or static taint analysis of Node.js packages [3]–[5], [33]–[35]. Ichnaea [33] uses

²In the paper where the user study results were presented [1], this value is referred to as *false positive rate*. However, based on the context of the paper and user survey questions, it corresponds to *precision* in our paper.

Jalangi2 program instrumentation [64] for dynamic taint analysis, tracking boolean taint values at runtime. AFFOGATO [35] performs taint propagation for string operations based on string similarity to achieve dynamic taint analysis, and Nodest [34] performs static taint analysis with abstract interpretation.

NODEMEDIC [3] uses Jalangi2 to perform dynamic taint analysis and generates a provenance graph as a witness of an uncovered tainted flow. NODEMEDIC-FINE [4] extends NODEMEDIC [3] with components that increase the number of flows found and vulnerabilities confirmed. Our work focuses on minimizing false negatives after taint analysis has already been performed. To that end, we leverage NODEMEDIC-FINE's provenance graph taint analysis output, and use GNN and LLM embedding layers for a final exploitability prediction. However, our work could be combined with any taint analysis tool that can generate provenance graphs.

B. ML-based vulnerability detection

Static and dynamic techniques have been extensively studied for uncovering security vulnerabilities [65]–[67]. Some examples include code-similarity-based methods [22], [68], [69], which detect vulnerabilities incurred by code cloning, and pattern-based methods [70], [71], which use rules to identify vulnerability patterns. However, both code cloning and pattern-based methods require human experts for final confirmation or to define the initial patterns. Our work focuses on using machine learning to reduce the dependency on human experts' annotations. Although there are no direct comparisons to our work (i.e., using building ML classifiers for exploitability on top of dynamic taint analysis), we list the most recent advances in ML-based vulnerability detectors.

Devign [72], IVDetect [59], and LineVD [73] used GNNs on a program's AST to classify a code block's vulnerability. LineVul [73] is the first ML-based predictor that used attention layers of a language model. Melicher et al. [74] train a DNN over JavaScript function AST features marked as vulnerable by a taint analysis that detects cross-site-scripting vulnerabilities. Neutaint [75] leverages saliency maps to predict the influence of input sources for tainted flows' sinks. DeepDFA [24] used data flow analysis to train a GNN and performed vulnerability detection more efficiently than prior program analysis based tools.

C. LLM-based vulnerability analysis

LLMs have been increasingly leveraged to support vulnerability detection and triage, spanning fine-grained localization, static-analysis enhancement, and false-positive reduction. LLMAO [76] is the first technique that finetunes a LLM to perform line-level (as opposed to file level or method level) vulnerability localization. MSIVD [25] built on top of DeepDFA by further finetuning the LLM embedding layers on a self-instruct augmented vulnerability dataset. LLift [77] integrates post-constraint guidance from an LLM into static analysis, enabling the discovery of new bugs in the Linux kernel. LLM4SA [78] uses program dependency analysis for code snippet extraction and LLM reasoning to triage thousands

of static bug warnings at scale in embedded operating systems and open-source C/C++ projects. Mohajer et al. [79] empirically studied ChatGPT’s ability to detect null dereferences and resource leaks in Java programs and to prune false positives from Infer’s warnings, showing improved precision.

Our work is the first to study the efficacy of using traditional ML and the most recent LLMs as classifiers on the actual exploitability of vulnerabilities discovered through dynamic taint analysis in Node.js packages.

VII. CONCLUSION

Our study demonstrates that combining machine learning with program analysis can enhance the results of ACE and ACI vulnerability triage in Node.js packages. Our findings show that applying machine learning to outputs from existing program analysis or vulnerability detection tools, or fine-tuning large language models for this task, can significantly reduce the fraction of benign packages having to be manually reviewed while maintaining a low risk of missing vulnerabilities. Future work could explore the co-design of program analysis tools and machine learning techniques—particularly advanced deep learning models—as auxiliary components to further enhance their capability to identify vulnerabilities more effectively and accurately.

VIII. ETHICS CONSIDERATIONS

Responsible disclosure. We follow a coordinated vulnerability disclosure process (i.e., responsible disclosure) [80]. We are in the process of responsibly disclosing vulnerabilities discovered in our evaluation to developers, with over 700 reported so far.

REFERENCES

- [1] M. Christakis and C. Bird, “What developers want and need from program analysis: an empirical study,” in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 332–343.
- [2] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.
- [3] D. Cassel, W. T. Wong, and L. Jia, “Nodemedic: End-to-end analysis of node.js vulnerabilities with provenance graphs,” in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023, pp. 1101–1127.
- [4] D. Cassel, N. Sabino, R. Martins, and L. Jia, “Nodemedic-fine: Automatic detection and exploit synthesis for node.js vulnerabilities.”
- [5] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, “Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1059–1076.
- [6] S. Lekies, B. Stock, and M. Johns, “25 million flows later: large-scale detection of dom-based xss,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1193–1204.
- [7] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, “Dexterjs: robust testing platform for dom-based xss vulnerabilities,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 946–949.
- [8] S. Bensalim, D. Klein, T. Barber, and M. Johns, “Talking about my generation: Targeted dom-based xss exploit generation using dynamic data flow analysis,” in *Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 27–33.
- [9] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz, “Towards automated generation of exploitation primitives for web browsers,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 300–312.
- [10] Y. Frempong, Y. Snyder, E. Al-Hossami, M. Sridhar, and S. Shaikh, “Hijax: Human intent javascript xss generator.” in *SECRYPT*, 2021, pp. 798–805.
- [11] M. Steffens and B. Stock, “Pmforce: Systematically analyzing postmessage handlers at scale,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 493–505.
- [12] T. M. Corporation, “CWE - CWE-94: Improper Control of Generation of Code (‘Code Injection’) (4.3),” 2020–, <https://cwe.mitre.org/data/definitions/94.html>.
- [13] ———, “CWE - CWE-77: Improper Neutralization of Special Elements used in a Command (‘Command Injection’) (4.3),” 2020–, <https://cwe.mitre.org/data/definitions/77.html>.
- [14] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltiformaggio, and W. Lee, “Towards measuring supply chain attacks on package managers for interpreted languages.”
- [15] D. Jang, R. Jhala, S. Lerner, and H. Shacham, “An empirical study of privacy-violating information flows in javascript web applications,” in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 270–283.
- [16] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, “An empirical study of information flows in real-world javascript,” in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2019, pp. 45–59.
- [17] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, “What are weak links in the npm supply chain?” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 331–340.
- [18] S. Bensalim, D. Klein, T. Barber, and M. Johns, “Talking about my generation: Targeted dom-based xss exploit generation using dynamic data flow analysis,” in *Proceedings of the 14th European Workshop on Systems Security*, 2021.
- [19] Y. Frempong., Y. Snyder., E. Al-Hossami., M. Sridhar., and S. Shaikh., “Hijax: Human intent javascript xss generator,” in *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT*, 2021.
- [20] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz, “Towards automated generation of exploitation primitives for web browsers,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.
- [21] M. Fu and C. Tantithamthavorn, “Linevul: A transformer-based line-level vulnerability prediction,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.
- [22] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “VulDeePecker: A deep learning-based system for vulnerability detection,” *arXiv preprint arXiv:1801.01681*, 2018.
- [23] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, “Automated vulnerability detection in source code using deep representation learning,” in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.
- [24] B. Steenhoek, H. Gao, and W. Le, “Dataflow analysis-inspired deep learning for efficient vulnerability detection,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
- [25] A. Z. Yang, H. Tian, H. Ye, R. Martins, and C. L. Goues, “Security vulnerability detection with multitask self-instructed fine-tuning of large language models,” *arXiv preprint arXiv:2406.05892*, 2024.
- [26] Mozilla, “JavaScript eval function documentation,” 2022, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval.
- [27] ———, “JavaScript Function constructor documentation,” 2022, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function.
- [28] Node.js, “Node.js exec API documentation,” 2022, https://nodejs.org/api/child_process.html#child_processexeccommand-options-callback.
- [29] ———, “Node.js execSync API documentation,” 2022, https://nodejs.org/api/child_process.html#child_processexecsynccommand-options.

- [30] C.-A. Staicu, M. Pradel, and B. Livshits, “SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS,” in *NDSS*, 2018.
- [31] T. M. Corporation, “CWE-22: Improper Limitation of a Pathname to a Restricted Directory (‘Path Traversal’),” 2020–, <https://cwe.mitre.org/data/definitions/22.html>.
- [32] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, “A Survey of Dynamic Analysis and Test Generation for JavaScript,” *ACM Computing Surveys*, 2017. [Online]. Available: <https://doi.org/10.1145/3106739>
- [33] R. Karim, F. Tip, A. Sochurkova, and K. Sen, “Platform-Independent Dynamic Taint Analysis for JavaScript,” *IEEE Transactions on Software Engineering*, 2018.
- [34] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, “Nodest: Feedback-driven static analysis of Node.js applications,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.
- [35] F. Gauthier, B. Hassanshahi, and A. Jordan, “AFFOGATO: Runtime detection of injection attacks for Node.js,” in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018.
- [36] S. Li, M. Kang, J. Hou, and Y. Cao, *Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis*, 2021.
- [37] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, “TaintPipe: Pipelined symbolic taint analysis,” in *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association.
- [38] N. Patnaik and S. Sahoo, “Javascript static security analysis made easy with JSPrime,” in *Blackhat USA*, 2013.
- [39] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, “Extracting Taint Specifications for JavaScript Libraries,” in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.
- [40] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, “Vcfinder: Finding potential vulnerabilities in open-source projects to assist code audits,” in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 426–437.
- [41] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, “The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches,” *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.
- [42] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, “Software vulnerability detection using deep neural networks: A survey,” *Proceedings of the IEEE*, vol. 108, pp. 1825–1848, 2020. [Online]. Available: <https://api.semanticscholar.org/CorpusID:222097224>
- [43] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?” *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.
- [44] G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai, “Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning,” *Journal of Systems and Software*, vol. 212, p. 112031, 2024.
- [45] J. Huang and K. C.-C. Chang, “Towards reasoning in large language models: A survey,” in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 1049–1065. [Online]. Available: <https://aclanthology.org/2023.findings-acl.67/>
- [46] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” *arXiv preprint arXiv:2106.09685*, 2021.
- [47] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “Qlora: Efficient finetuning of quantized LLMs,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [48] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [49] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks,” *arXiv preprint arXiv:1511.05493*, 2015.
- [50] “Fast implementation,” available at: <https://github.com/fast-sp-2023/fast>.
- [51] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [52] D. G. DeepSeek-AI, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning,” *arXiv preprint arXiv:2501.12948*, 2025.
- [53] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, “The llama 3 herd of models,” *arXiv preprint arXiv:2407.21783*, 2024.
- [54] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu, “Qwen2.5 technical report,” *arXiv preprint arXiv:2412.15115*, 2024.
- [55] “Hugging face models,” available at: https://huggingface.co/models?pipeline_tag=text-generation&sort=downloads.
- [56] G. Comanici, E. Bieber, M. Schaekermann, I. Pasupat, N. Sachdeva, I. Dhillon, M. Blstein, O. Ram, D. Zhang, E. Rosen *et al.*, “Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities,” *arXiv preprint arXiv:2507.06261*, 2025.
- [57] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [58] “Openrouter models,” available at: <https://openrouter.ai>.
- [59] Y. Li, S. Wang, and T. N. Nguyen, “Vulnerability detection with fine-grained interpretations,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.
- [60] P. S. Kochhar, X. Xia, D. Lo, and S. Li, “Practitioners’ expectations on automated fault localization,” in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 165–176.
- [61] J. Cohen, “A coefficient of agreement for nominal scales,” *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.
- [62] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774. [Online]. Available: <http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf>
- [63] Y. Mao, Y. Ge, Y. Fan, W. Xu, Y. Mi, Z. Hu, and Y. Gao, “A survey on lora of large language models,” *Frontiers of Computer Science*, vol. 19, no. 7, p. 197605, 2025.
- [64] K. Sen, S. Kalsapur, T. Brutch, and S. Gibbs, “Jalangi: A selective record-replay and dynamic analysis framework for JavaScript,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.
- [65] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.
- [66] H. Shahriar and M. Zulkernine, “Mitigating program security vulnerabilities: Approaches and challenges,” *ACM Comput. Surv.*, vol. 44, no. 3, jun 2012. [Online]. Available: <https://doi.org/10.1145/2187671.2187673>
- [67] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.
- [68] S. Kim, S. Woo, H. Lee, and H. Oh, “Vuddy: A scalable approach for vulnerable code clone discovery,” in *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017, pp. 595–614.
- [69] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, “Vulpecker: an automated vulnerability detection system based on code similarity analysis,” in *Proceedings of the 32nd annual conference on computer security applications*, 2016, pp. 201–213.
- [70] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 529–540.
- [71] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: Exposing missing checks in source code for vulnerability discovery,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 499–510.
- [72] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics

- via graph neural networks,” *Advances in neural information processing systems*, vol. 32, 2019.
- [73] D. Hin, A. Kan, H. Chen, and M. A. Babar, “Linevd: Statement-level vulnerability detection using graph neural networks,” in *Proceedings of the 19th international conference on mining software repositories*, 2022, pp. 596–607.
- [74] W. Melicher, C. Fung, L. Bauer, and L. Jia, “Towards a lightweight, hybrid approach for detecting dom xss vulnerabilities with machine learning,” in *Proceedings of the Web Conference 2021*, ser. WWW ’21, 2021.
- [75] D. She, Y. Chen, B. Ray, and S. Jana, “Neutaint: Efficient dynamic taint analysis with neural networks,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.
- [76] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, “Large language models for test-free fault localization,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.
- [77] H. Li, Y. Hao, Y. Zhai, and Z. Qian, “Enhancing static analysis for practical bug detection: An llm-integrated approach,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.
- [78] C. Wen, Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and T. Cong, “Automatically inspecting thousands of static bug warnings with large language model: How far are we?” *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 7, pp. 1–34, 2024.
- [79] M. M. Mohajer, R. Aleithan, N. S. Harzevili, M. Wei, A. B. Belle, H. V. Pham, and S. Wang, “Effectiveness of chatgpt for static analysis: How far are we?” in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 2024, pp. 151–160.
- [80] CERT, “The CERT guide to coordinated vulnerability disclosure,” 2023, <https://vuls.cert.org/confluence/display/CVD>.

APPENDIX A FULL EXPERIMENTAL RESULTS

We present the complete results of our experiments on classical machine learning and LLM methods in Table VIII, whereas Table IV reports only the metrics for the best-performing setups. The table includes the mean and variance of F1, precision, recall, and accuracy for each model across five runs.

TABLE VIII: F1 Score (F1), Precision (Prec), Recall (Rec), and Accuracy (Acc) for all classical ML methods and LLM methods. Higher metrics indicate better performance. All metrics are reported as the mean and variance over five runs with different random seeds.

Model	F1		Prec		Rec		Acc	
	Mean	Var	Mean	Var	Mean	Var	Mean	Var
Random Forest								
-attn	0.899	1.83e-05	0.914	5.41e-05	0.885	9.06e-05	0.856	3.36e-05
-avg	0.900	8.37e-06	0.917	1.99e-07	0.883	2.66e-05	0.857	1.40e-05
-max	0.897	4.29e-06	0.922	9.58e-06	0.875	1.07e-05	0.855	8.40e-06
XGBoost								
-attn	0.898	4.76e-05	0.914	1.01e-04	0.882	1.44e-04	0.854	9.24e-05
-avg	0.904	0.00e+00	0.917	0.00e+00	0.891	0.00e+00	0.862	0.00e+00
-max	0.898	0.00e+00	0.930	0.00e+00	0.869	0.00e+00	0.857	0.00e+00
Logistic Regression								
-attn	0.882	9.42e-05	0.876	2.72e-04	0.888	6.93e-05	0.828	2.32e-04
-avg	0.856	0.00e+00	0.844	0.00e+00	0.869	0.00e+00	0.788	0.00e+00
-max	0.892	0.00e+00	0.909	0.00e+00	0.876	0.00e+00	0.847	0.00e+00
SVM								
-attn	0.869	7.44e-05	0.850	6.84e-05	0.888	3.09e-04	0.805	1.32e-04
-avg	0.857	0.00e+00	0.839	0.00e+00	0.876	0.00e+00	0.788	0.00e+00
-max	0.898	0.00e+00	0.898	0.00e+00	0.898	0.00e+00	0.852	0.00e+00
OpenAI o4-mini-high								
-zero-shot	0.805	7.49e-05	0.889	5.64e-05	0.736	2.77e-04	0.742	7.53e-05
DeepSeek R1-0528								
-zero-shot	0.857	5.70e-05	0.824	1.04e-04	0.893	3.36e-04	0.784	1.03e-04
Gemini 2.5 Pro								
-zero-shot	0.851	1.85e-04	0.769	3.95e-04	0.953	1.49e-04	0.758	6.08e-04
OpenAI GPT-4.1								
-zero-shot	0.858	9.20e-06	0.778	7.45e-06	0.956	2.66e-05	0.770	2.25e-05
DeepSeek-R1-Distill-Qwen-14B								
-zero-shot	0.816	7.41e-05	0.803	9.35e-05	0.831	2.77e-04	0.729	1.32e-04
-linear-probe	0.788	3.16e-04	0.744	3.13e-04	0.838	2.20e-03	0.673	4.26e-04
-lora-ft	0.866	1.29e-04	0.861	2.90e-04	0.872	2.29e-04	0.804	2.94e-04
DeepSeek-R1-Distill-Llama-8B								
-zero-shot	0.793	8.14e-05	0.721	1.23e-04	0.882	1.70e-04	0.667	2.38e-04
-linear-probe	0.773	8.71e-04	0.729	9.67e-05	0.826	4.01e-03	0.651	9.52e-04
-lora-ft	0.889	2.70e-04	0.878	1.93e-04	0.901	5.75e-04	0.837	5.37e-04
-full	0.684	1.47e-01	0.663	1.41e-01	0.715	1.68e-01	0.679	5.39e-02
-full+GNN	0.551	1.48e-01	0.692	1.53e-01	0.523	1.68e-01	0.582	5.44e-02
DeepSeek-R1-Distill-Qwen-7B								
-zero-shot	0.723	8.41e-04	0.763	4.75e-04	0.688	1.74e-03	0.619	1.12e-03
-linear-probe	0.779	5.40e-04	0.739	1.14e-04	0.823	2.28e-03	0.661	7.14e-04
-lora-ft	0.845	2.81e-04	0.834	9.61e-04	0.857	3.62e-04	0.771	7.61e-04
-full	0.915	1.72e-04	0.900	1.90e-04	0.931	2.56e-04	0.875	3.72e-04
-full+GNN	0.912	1.25e-04	0.890	3.74e-05	0.934	3.46e-04	0.869	2.44e-04
Llama-3.1-8B-(Instruct)								
-zero-shot	0.303	1.00e-03	0.709	4.06e-03	0.193	4.42e-04	0.358	6.80e-04
-linear-probe	0.794	3.62e-04	0.740	3.17e-04	0.857	1.05e-03	0.678	7.89e-04
-lora-ft	0.909	9.46e-05	0.888	6.67e-05	0.930	2.29e-04	0.865	1.90e-04
-full	0.505	1.61e-01	0.667	1.53e-01	0.530	2.32e-01	0.539	4.76e-02
-full+GNN	0.831	5.58e-04	0.849	3.62e-04	0.816	2.91e-03	0.761	6.49e-04
Qwen2.5-Coder-14B-(Instruct)								
-zero-shot	0.679	2.70e-04	0.779	4.96e-04	0.602	3.09e-04	0.587	4.62e-04
-linear-probe	0.729	1.62e-03	0.743	1.49e-04	0.718	6.20e-03	0.616	1.46e-03
-lora-ft	0.863	1.18e-04	0.846	8.87e-05	0.880	4.95e-04	0.797	2.04e-04
Qwen2.5-Coder-7B-(Instruct)								
-zero-shot	0.526	4.19e-04	0.782	4.19e-04	0.397	4.42e-04	0.483	2.72e-04
-linear-probe	0.744	5.26e-04	0.745	8.12e-04	0.746	2.97e-03	0.630	7.42e-04
-lora-ft	0.859	3.57e-04	0.859	4.99e-04	0.860	3.57e-04	0.796	7.92e-04
-full	0.896	6.17e-04	0.893	5.91e-05	0.901	2.68e-03	0.850	9.88e-04
-full+GNN	0.902	6.26e-05	0.882	4.27e-04	0.923	3.89e-04	0.854	1.62e-04