A Sequential Multi-Stage Approach for Code Vulnerability Detection via Confidence- and Collaboration-based Decision Making

Chung-Nan Tsai[1], Xin Wang[2], Cheng-Hsiung Lee[3] and Ching-Sheng Lin[3,*]

[1]Lam Research Japan GK, Japan
[2]College of Integrated Health Sciences and the AI Plus Institute, University at Albany, USA
[3]Master Program of Digital Innovation, Tunghai University, Taiwan
chung-nan.tsai@lamresearch.com, xwang56@albany.edu, {hsiung, cslin612}@thu.edu.tw

Abstract

While large language models (LLMs) have shown strong capabilities across diverse domains, their application to code vulnerability detection holds great potential for identifying security flaws and improving software safety. In this paper, we propose a sequential multi-stage approach via confidence- and collaboration-based decision making (ConColl). The system adopts a three-stage sequential classification framework, proceeding through a single agent, retrieval-augmented generation (RAG) with external examples, and multi-agent reasoning enhanced with RAG. The decision process selects among these strategies to balance performance and cost, with the process terminating at any stage where a high-certainty prediction is achieved. Experiments on a benchmark dataset and a low-resource language demonstrate the effectiveness of our framework in enhancing code vulnerability detection performance.

1 Introduction

As AI technologies continue to grow in popularity, with many projects developed in open-source environments, ensuring software quality and effectively identifying vulnerabilities has become increasingly important (Harzevili et al., 2024; Islam et al., 2024). To tackle vulnerability detection, researchers have applied machine learning and deep learning to identify security flaws in source code. While these approaches have achieved promising performance, most of them depend on either fine-tuning pre-trained models or training neural networks from scratch (Zhou et al., 2024). Large language models (LLMs) have recently achieved remarkable success across a wide spectrum of tasks. Building on this progress, LLMs have emerged as a powerful tool for vulnerability detection and present a more promising alternative to traditional learning-based approaches (Shestov et al., 2025). However, despite their effectiveness, deploying LLMs for vulnerability detection in a practical setting raises significant concerns regarding efficiency, scalability and cost (Liu et al., 2025).

In the context of LLM-based vulnerability detection, a cost-effective detection framework should aim to make accurate predictions with minimal computational and contextual overhead. Prior works have focused on optimizing static prompts (Zhang et al., 2023) or leveraging multiple LLMs of varying sizes to reduce computational cost (Chen et al., 2023). Nevertheless, these approaches may be impractical due to deployment complexity, model compatibility, and resource constraints. In this paper, we explore a cost-effective alternative using a single LLM, where inference pathways are adaptively chosen based on internal confidence and collaborative signals. By progressively increasing reasoning complexity—from direct single agent prediction, to retrieval-augmented generation (RAG) with few samples, and finally to multi-agent collaboration—we enable efficient vulnerability detection while maintaining strong performance. This design avoids multi-LLM overhead and ensures consistent representations, while remaining generalizable to existing LLM-based detection pipelines.

The main contributions of this paper can be outlined as follows.

- We propose a single-LLM architecture that dynamically selects among direct classification by a single agent, retrieval-augmented prediction, and multi-agent collaboration. The selection is guided by the

* Corresponding author

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 21151–21157
November 4-9, 2025

model's internal confidence and collaborative signals. This adaptive pipeline improves cost-efficiency by applying simple reasoning to easy cases and reserving more complex methods for harder ones.

- Experiments on the TreeVul_Ext benchmark dataset suggest that our method can achieve competitive detection performance while reducing computational overhead (Zhou et al., 2024). The framework also shows potential in handling low-resource languages and indicating its practicality in real-world applications (Le et al., 2024).

2 Related Work

In current machine learning applications for code vulnerability detection, deep learning architectures have become the dominant paradigm. For example, VulDeePecker employs a BiLSTM architecture to automatically learn vulnerability patterns from code gadgets without relying on manually crafted features (Li et al., 2018). Some approaches adopt hybrid architectures, such as CNN combined with Random Forest (Russell et al., 2018) and CNN-BiLSTM (Gu et al., 2025), to leverage both syntactic and semantic feature extraction capabilities. Transformer architecture, originally designed for natural language processing, has proven highly effective in modeling context semantics (Vaswani et al., 2017). This architecture has been widely adopted in software vulnerability detection tasks. Notable Transformer-based models used in this domain include CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and CodeBERTa (Karmakar et al., 2021). CodeBERT and CodeBERTa learn joint representations of source code and natural language, while GraphCodeBERT extends this approach by incorporating data-flow information derived from abstract syntax trees.

LLMs have achieved remarkable success across a wide range of domains. Examples include OpenAI's ChatGPT and Meta's LLaMA, illustrating rapid LLM progress. To further enhance performance, techniques such as RAG and prompt engineering are commonly adopted. With no exception to their demonstrated success in various fields, LLMs have also been applied to software vulnerability detection. Existing methods include both single-model and mixture-of-LLMs approaches, often leveraging RAG for few-shot context and prompt engineering to elicit stronger reasoning from models (Widyasari et al., 2024). Role-based simulation in LLMs has emerged as a widely adopted technique to enhance task-specific performance and decision-making (White et al., 2023). MuCoLD assigns developers and security testers roles to LLMs, so that they can collaboratively identify software vulnerabilities through multi-round discussions (Mao et al., 2024).

3 Methodology

3.1 Problem Description

This paper focuses on function-level binary classification (Kluban et al., 2022; Fu et al., 2024), where the goal is to determine whether a given function contains security vulnerabilities. We formulate the detection task as a mapping $f : (X,Z;\theta) \rightarrow Y$, where X represents the input source code, Z denotes optional auxiliary information that may be required by the model, θ indicates the model parameters and Y is the binary label with 1 for vulnerable cases and 0 for non-vulnerable ones. In this study, we investigate vulnerability detection using a single LLM with fixed parameters θ, while optionally incorporating additional information Z, such as crafted prompts from prompt engineering and retrieved evidence from RAG.

3.2 Proposed Framework

In this section, we present the sequential multi-stage approach where the system architecture is described in the Figure 1. Our main confidence- and collaboration-based decision module (ConColl) is composed of three stages, each progressively incorporating more reasoning capability to handle increasingly ambiguous cases.

Stage 1: Direct Prediction with Single LLM. At this stage, we employ a single LLM with a task-specific prompt (shown below) to perform binary classification on the input source code. The model is prompted to analyze the code and respond with either "Yes" or "No" as the first generated token. We define the confidence score (C.S.) as the difference in predicted probabilities between the top-1 and top-2 tokens at this initial position. If the score exceeds a predefined threshold (th1) and the top-1 token is either "Yes" or "No", the prediction is accepted. Otherwise, the decision is deferred to a more elaborate subsequent stage.

$Prompt_{single}$: Is the following code vulnerable? Respond with only 'Yes' or 'No'. \n\n {Source_code}

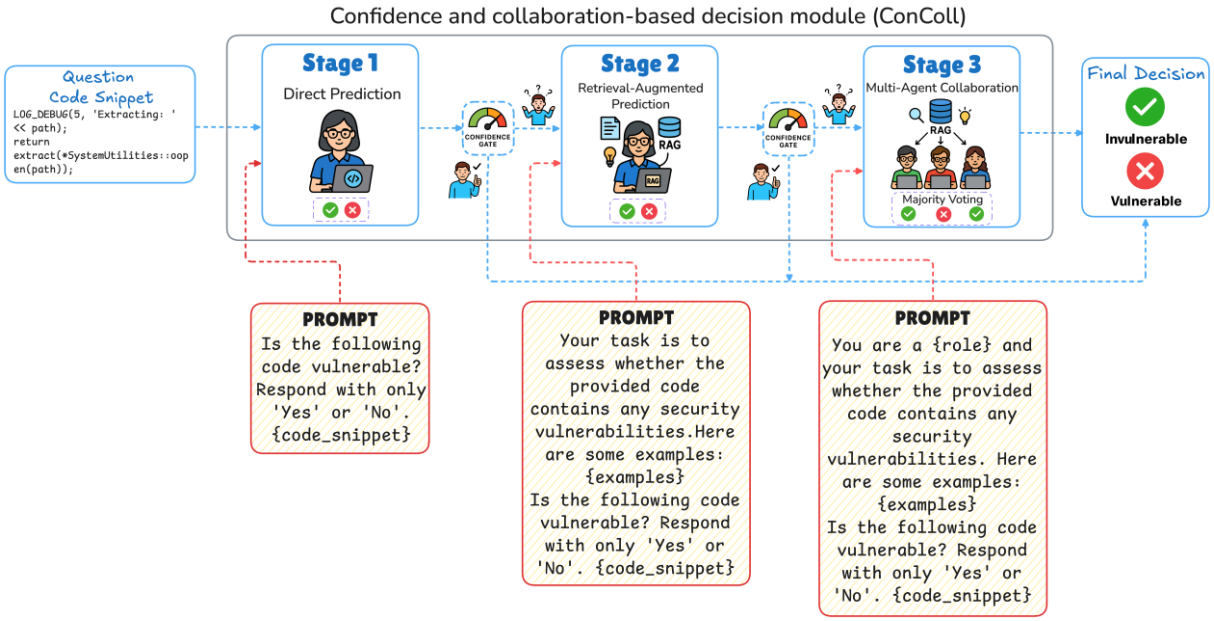

Figure 1: The system architecture.

Stage 2: Retrieval-Augmented Prediction. If the model fails to make a confident prediction in the first stage, the system proceeds to a second stage for further reasoning. In this stage, we retrieve several semantically relevant positive and negative examples from the training dataset based on the input source code. These examples, along with the original input, are then provided to the language model to support its inference. The prompt used at this stage is shown below. Similar to the first stage, we compute the C.S. based on the probability distribution over the model's predicted tokens. If the score exceeds a second predefined threshold (th2) and the top-1 prediction corresponds to a valid class label ("Yes" or "No"), the model's decision is approved. Otherwise, the system is routed to Stage 3 for further analysis.

$Prompt_{RAG}$: Your task is to assess whether the provided code contains any security vulnerabilities. \n\n Here are some examples: \n\n {examples} \n\n Is the following code vulnerable? Respond with only 'Yes' or 'No'. \n\n {Source_code}

Stage 3: Multi-Agent Collaboration. If the model still lacks sufficient confidence after Stage 2, the system proceeds to Stage 3, which employs a multi-agent decision mechanism. In this stage, multiple instances of the language model are activated, each operating under slightly different reasoning roles. These agents are provided with the retrieved reference samples from Stage 2 as contextual support. Each agent generates a prediction based on its role-specific prompt. The final decision is made through majority voting across all agent outputs. We adopt three role settings (security analyst, penetration tester and software security engineer) and use the following prompts.

$Prompt_{agent}$: You are a {role} and your task is to assess whether the provided code contains any security vulnerabilities. \n\n Here are some examples: \n\n {examples} \n\n Is the following code vulnerable? Respond with only 'Yes' or 'No'. \n\n {Source_code}

The complete process of our method is displayed in Appendix A. Our LLM treats the test code, the prompt, and relevant examples (if needed) as the input for testing. The *RetrieveRel* function takes the test code along with the training data to extract relevant examples. The *Voting* function aggregates the predictions from three agents and produces the final result through majority voting.

4 Experiments

4.1 Dataset and Evaluation Metrics

We conduct our experiments using the publicly available TreeVul_Ext dataset, which comprises 20 open-source C/C++ software repositories (Zhou et al., 2024). The dataset is composed of 7,683 training functions, 853 validation functions, and 386 test functions. Notably, we do not perform any training or fine-tuning on the model. Instead, we only adopt the RAG approach that retrieves

semantically similar samples from the training set to the target function under analysis.

We compare our model against several baseline models using standard evaluation metrics, including precision (Pre), recall (Rec), F1-score (F1), and accuracy (Acc). These metrics provide a rigorous basis for comparing the predictive performance across models.

4.2 Performance Evaluation

In this section, we compare our proposed model with two categories of baseline methods on the TreeVul_Ext. The first category includes LLM-based approaches, such as GPT-3.5, LLaMA-3-8B, Gemma-7B, Mixtral-8x7B, GPT-4o, and an ensemble voting method that aggregates their predictions (Widyasari et al., 2024). The second category consists of training-based models, including CodeBERT, CodeBERTa, and GraphCodeBERT. The implementation details are reported in Appendix B.

Model	Acc	Pre	Rec	F1
GPT-3.5	62.7	**76.3**	36.8	49.7
LLaMA-3-8B	60.9	63.5	51.3	56.7
Gemma-7B	67.6	71.8	58.0	64.2
Mixtral-8x7B	63.2	73.4	41.5	53.0
GPT-4o	67.4	66.8	68.9	67.9
Ensemble Voting	**68.4**	75.2	54.9	63.5
CodeBERT	60.3	62.3	53.3	57.3
CodeBERTa	61.7	57.8	86.0	69.2
GraphCodeBERT	59.0	56.9	74.6	64.5
Our Model	62.9	57.7	**96.8**	**72.3**

Table 1: Experimental results of different models (%).

As shown in Table 1, our model outperforms all compared baselines in terms of F1-score. Among the LLM-based approaches, GPT-4o achieves the best performance, while CodeBERTa leads among training-based methods. Our model surpasses GPT-4o by 6.5% and CodeBERTa by 4.5% in F1-score. Our model demonstrates a strong recall-oriented behavior and achieves a recall as high as 96.8%. This suggests that the model is highly sensitive to potential vulnerabilities, which makes it particularly effective in minimizing false negatives. In contrast, GPT-3.5 achieves the highest precision among all methods (76.3%). However, it also suffers from the lowest recall, indicating a tendency to produce more conservative predictions while missing a larger number of actual vulnerabilities. Meanwhile, the ensemble voting approach obtains the highest accuracy among all methods (68.4%), although its F1-score remains lower than that of certain individual LLMs (i.e., Gemma-7B and GPT-4o).

Compared to single-pass inference using a standalone LLM, our sequential framework introduces additional latency due to its multi-stage decision process. Specifically, the later stages, such as retrieval-based reference sample generation and multi-agent collaboration, involve multiple API calls, which result in longer inference times. While training-based methods may require significant upfront computational costs, they often benefit from faster runtime predictions, highlighting a trade-off between adaptability and efficiency. Several representative prediction cases are presented Appendix C.

Model	F1	Time	Cost
Our Model	72.3%	6:15	$0.66
Stage1	70.4%	3:28	$0.15
Stage2	71.3%	4:20	$0.40
Stage3	72.5%	10:56	$1.13

Table 2: Performance of each stage vs. full model.

To evaluate the effectiveness and efficiency of our proposed method, we conduct a study comparing the individual stages with the complete sequential model. The comparison includes performance metrics as well as time and cost expenditures. As shown in Table 2, while Stage 3 individually achieves the highest F1-score (72.5%), it also requires the longest runtime (10:56) and the highest computational cost ($1.13). This suggests that although Stage 3 is the most effective in isolation, it may not be practical in resource-constrained scenarios. In comparison, our sequential model integrates all stages and achieves a near-optimal F1-score (72.3%) while keeping time and cost at moderate levels (6:15 and $0.66). This indicates that the sequential design effectively balances performance and efficiency.

Model	Pre	Rec	F1
GPT Fine-Tuning	0.37	0.32	0.34
GPT Few-Shot	**0.43**	0.44	0.43
CodeBERT	0.24	0.47	0.32
Our Model	0.31	**0.73**	**0.44**

Table 3: Evaluation results on the Kotlin dataset.

To test the applicability, we apply the model to a low-resource programming language where Kotlin serves as the test case. The dataset consists of 20 vulnerable functions and 98 non-vulnerable functions (Le et al., 2024). Given the small data size and class imbalance, we adopt a 10-round evaluation strategy. In each round, the data is randomly split into 60% for training, 20% for validation, and 20% for testing. Final metrics are

averaged over all rounds to ensure evaluation reliability (shown in Table 3). Consistent with the results on the TreeVul_Ext dataset, our model achieves the best F1-score among all compared methods, with recall remaining the most outstanding. These results indicate the potential of our sequential framework to generalize effectively, even under low-resource conditions.

5 Conclusion and Future Work

This paper presents a preliminary exploration of a sequential decision framework for cost-effective software vulnerability detection using LLMs. Our framework incrementally increases reasoning complexity through three stages—direct prediction, retrieval-augmented prediction, and multi-agent collaboration—based on model confidence to balance quality and cost. The results on both benchmark datasets suggest that our method is a potential alternative to existing models.

Although our model achieves promising results, there remain several avenues for further improvement. First, an adaptive confidence thresholding mechanism can be explored to improve the efficiency and flexibility of stage transitions. Second, adding static analysis or program semantics may improve detection of certain vulnerabilities Lastly, the proposed framework can be generalized to support cross-language and cross-task applicability.

Limitations

This study has several limitations. First, our approach has not been evaluated on large-scale datasets, and future work is needed to assess its scalability. Second, we used manually constructed prompts without applying systematic optimization, potentially leaving room for performance improvements through prompt engineering. Third, all experiments were conducted using ChatGPT, and the effectiveness of the proposed method with other large language models remains to be explored.

AI Assistants in Research or Writing

We used ChatGPT to correct grammatical errors and polish the language.

References

Lingjiao Chen, Matei Zaharia, and James Zou. 2023. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and others. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2024. AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering* 29, 1, 4.

Wanyi Gu, Guojun Wang, Peiqiang Li, Guangxin Zhai, and Xubin Li. 2025. Detecting unknown vulnerabilities in smart contracts with the CNN-BiLSTM model. *International Journal of Information Security* 24, 33.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, and others. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming Jiang, and Nachiappan Nagappan. 2024. A systematic literature review on automated software vulnerability detection using machine learning. *ACM Computing Surveys* 57, 3, 1–36.

Nafis Tanveer Islam, Gonzalo De La Torre Parra, Dylan Manual, Murtuza Jadliwala, and Peyman Najafirad. 2024. Causative Insights into Open Source Software Security using Large Language Code Embeddings and Semantic Vulnerability Graph. *arXiv preprint arXiv:2401.07035*.

Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE*, 1332–1336.

Maryna Kluban, Mohammad Mannan, and Amr Youssef. 2022. On measuring vulnerable javascript functions in the wild. In *Proceedings of the 2022 ACM on Asia conference on computer and communications security*, 917–930.

Triet Huynh Minh Le, M Ali Babar, and Tung Hoang Thai. 2024. Software vulnerability prediction in low-resource languages: An empirical study of codebert and chatgpt. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 679–685.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*.

Jianing Liu, Guanjun Lin, Huan Mei, Fan Yang, and Yonghang Tai. 2025. Enhancing vulnerability detection efficiency: An exploration of light-weight LLMs with hybrid code features. *Journal of Information Security and Applications* 88, 103925.

Zhenyy Mao, Jialong Li, Dongming Jin, Munan Li, and Kenji Tei. 2024. Multi-role consensus through llms discussions for vulnerability detection. In *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, IEEE, 1318–1319.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, IEEE, 757–762.

Aleksei Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Pavel Zadorozhny, Anton Cheshkov, Rustam Mussabayev, Alymzhan Toleu, Gulmira Tolegen, and Alexander Krassovitskiy. 2025. Finetuning large language models for vulnerability detection. *IEEE Access*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*.

Ratnadira Widyasari, David Lo, and Lizi Liao. 2024. Beyond ChatGPT: Enhancing Software Quality Assurance Tasks with Diverse LLMs and Validation Techniques. *arXiv preprint arXiv:2409.01001*.

Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879*.

Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 47–51.

A Algorithm

Algorithm 1: Confidence- and Collaboration-based Decision

Input: Training data $\{X^{train}, Y^{train}\}$, Testing data $\{X^{test}, Y^{test}\}$, Language model LLM, thresholds th1, th2.

Output: Predicted labels $\{\hat{y}^{test}\}$.

for each x_i^{test} in X^{test} do
  //Stage 1: Direct Prediction
  pred1, cs1=LLM(x_i^{test}, [$Prompt_{single}$, None])
  if cs1>=th1:
    $\hat{y}_i^{test}$=pred1
    continue
  end if
  //Stage 2: Retrieval-Augmented Prediction
  RAG_i=RetrieveRel(x_i^{test}, X^{train}, Y^{train})
  pred2, cs2=LLM(x_i^{test}, [$Prompt_{RAG}$, RAG_i])
  if cs2>=th2:
    $\hat{y}_i^{test}$=pred2
    continue
  end if
  //Stage 3: Multi-Agent Collaboration
  predA, _=LLM(x_i^{test}, [$Prompt_{agent}^{A}$, RAG_i])
  predB, _=LLM(x_i^{test}, [$Prompt_{agent}^{B}$, RAG_i])
  predC, _=LLM(x_i^{test}, [$Prompt_{agent}^{C}$, RAG_i])
  $\hat{y}_i^{test}$ = Voting(predA, predB, predC)
end for
return $\hat{y}^{test}$

B Implementation Details

All experiments are conducted on a Windows 10 operating system with an Intel Core i9 processor, 128 GB of RAM, and an NVIDIA GeForce RTX 3090 GPU with 24 GB of memory. For our approach, we employ OpenAI's gpt-3.5-turbo as the underlying LLM, whereas the training-based baseline models are implemented using the Hugging Face library.

C Examples

Table 4 presents three representative examples where the final predictions are correct. These examples respectively require one, two, and three stages of processing. For cases involving more than two stages, we additionally report the intermediate predictions at each stage. However, the associated confidence scores at these intermediate steps are not sufficiently reliable and should be interpreted with caution. For clarity, in the table, 1 denotes a vulnerable case and 0 denotes a non-vulnerable case in both the Target and Prediction columns.

Code snippet	Stage	Prediction	Target
LOG_DEBUG(5, 'Extracting: ' << path); return extract(*SystemUtilities::oopen(path));	1	1	1

if(flow->http.user_agent[0] != '\0')
fprintf(out, '[UserAgent: %s]', flow->http.user_agent);
}
	1	1	1

	2	1	

static PyObject* ast2obj_object(void *o)
{
if (!o)
o = Py_None;
Py_INCREF((PyObject*)o);
return (PyObject*)o;
}
	1	1	0

	2	1	
	3	0	

Table 4: Representative prediction examples.