LLM-Driven Taint Analysis for Detecting Bad Partitioning Issues in Trusted Applications

Tatsuya YOSHINAGA
Nihon University
Tokyo, Japan
cstt26007@g.nihon-u.ac.jp

Minami YODA
Nihon University
Tokyo, Japan
yoda.minami@nihon-u.ac.jp

Yutaka MATSUNO
Nihon University
Tokyo, Japan
matsuno.yutaka@nihon-u.ac.jp

***Abstract*—In Trusted Applications (TAs) in Trusted Execution Environments (TEEs), bad partitioning issues caused by poor trust-boundary design are an important problem. Existing rule-based static analysis is effective for these issues. However, it has difficulty detecting cases that require cross-function data propagation or context-dependent interpretation. This paper proposes an LLM-driven taint-analysis pipeline for detecting bad partitioning issues. The method first broadly screens candidate functions, then builds function chains, and performs three-stage analysis with START, MIDDLE, and END. In an evaluation on a single labeled TA, we ran the proposed method and DITING independently to compare their detection characteristics and to examine the effectiveness of prompt elements in the proposed method. The results show that the proposed method and DITING have complementary detection characteristics, and that including TEE-specific analysis conditions in the prompt strongly improves detection performance.**

***Index Terms*—Trusted Execution Environment, Trusted Application, Bad Partitioning Issues, Taint Analysis, Large Language Model**

I. Introduction

Trusted Execution Environments (TEEs) provide isolated execution for functions that require strong confidentiality and integrity. In a software TEE such as OP-TEE, Trusted Applications (TAs) interact with the untrusted Rich Execution Environment (REE) through shared memory and parameters, so data must be handled safely across the boundary. If this boundary is designed poorly, bad partitioning issues can occur, such as plain-text output, insufficient input validation, and unsafe use of shared memory [1].

Rule-based static taint analysis is effective for these problems, and DITING [1] is a representative example. However, rule-based methods alone have difficulty with code that requires cross-function data propagation or context-dependent interpretation. Recent surveys report growing interest in applying LLMs to software vulnerability detection [2], and SymTEE [3] has also been proposed for TEE. However, TEE-focused LLM work has mainly studied missing input validation [3], and LLM-based detection of broader bad partitioning issues remains less explored.

This paper proposes an LLM-driven taint-analysis pipeline for bad partitioning issues in TAs. In this pipeline, candidate functions are screened first, and then each function chain is analyzed through START, MIDDLE, and END for candidate generation and final judgment. To evaluate the detection characteristics of the proposed method and the effectiveness of prompt elements, we define two research questions. RQ1 asks what detection performance and detection characteristics the proposed method shows for detecting bad partitioning issues in TA compared with DITING. RQ2 asks which prompt elements in the proposed pipeline are effective for handling TEE-specific analysis conditions.

The main contributions of this paper are as follows:

1) We propose an LLM-driven taint-analysis pipeline for bad partitioning issues.
2) Through prompt comparisons and evaluation across multiple models, we show an effective prompt design and differences in complementary effects across models.
3) We quantitatively compare the complementarity of DITING and the LLM pipeline, showing that union recall improves, while gains in union F1 depend on false-positive control.

II. Background and Related Work

A. Bad Partitioning Issues

Ma et al. summarized bad partitioning issues caused by poor trust-boundary design in TEE into three categories. Unencrypted Data Output (UDO) occurs when secret data is written to shared memory without encryption. Input Validation Weakness (IVW) occurs when REE-derived parameters are used in memory operations without boundary checks. Direct Usage of Shared Memory (DUS) occurs when data in shared memory is used directly instead of being copied into a private region first [1].

B. Rule-Based and LLM-Based Detection

DITING [1] is a CodeQL-based rule-based static analysis tool that applies detection rules for UDO, IVW, and DUS to TAs. Ma et al. released both DITING and PartitioningE-Bench [1], [4], and we use a labeled TA from this benchmark in our evaluation. At the same time, more studies have applied LLMs to code vulnerability detection [2]. IRIS [5] and LATTE [6] combine LLMs with static analysis or taint tracking, while SymTEE [3] uses LLM-assisted symbolic execution to detect input-validation flaws in TAs. However, LLM-based detection of broader bad partitioning issues, including UDO and DUS, remains underexplored.

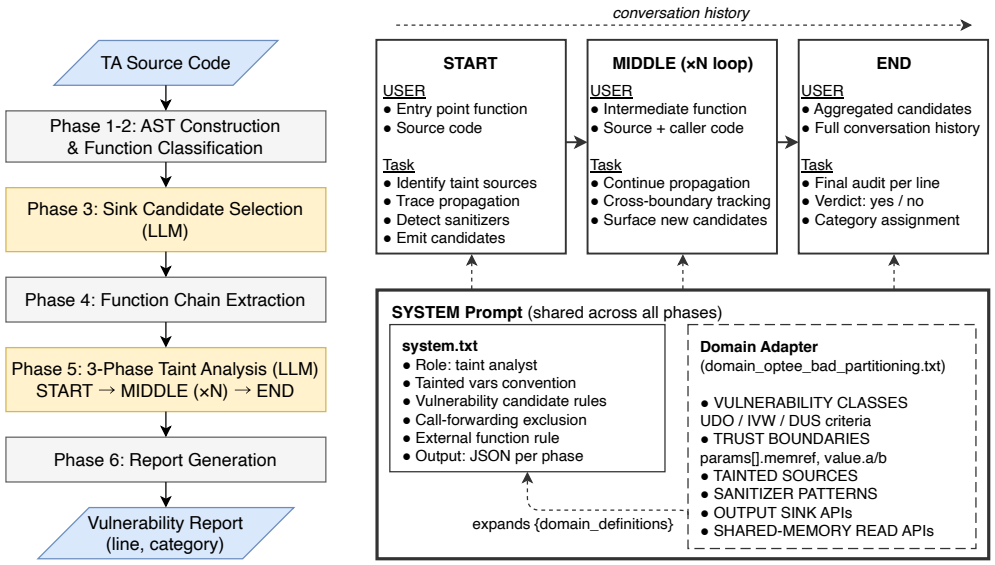

Fig. 1. Overview of the pipeline. Phases 1–4 extract function chains from TA source code. Phase 5 performs three-stage taint analysis with the domain adapter, and Phase 6 generates the final report.

III. Methodology

A. Pipeline Overview

Fig. 1 shows the overall LLM analysis pipeline used in this study. The system is inspired by LATTE's staged LLM-based taint-analysis workflow [6]. In Phases 1 and 2, we build an AST from TA source code and classify functions. In Phase 3, the LLM screens candidate functions that may act as sinks. In Phase 4, we extract function chains from the entry point to each candidate function. In Phase 5, we perform three-stage taint analysis consisting of START, MIDDLE, and END for each chain. In Phase 6, we generate the final report. DITING and our pipeline are executed independently, and we compare their detection results. In this method, Phase 3 is designed to keep the candidate set broad, while Phase 5 uses the domain adapter to judge whether TEE-specific conditions are satisfied.

B. Candidate Screening and Staged Taint Analysis

Below, we describe sink-candidate screening in Phase 3 and staged taint analysis in Phase 5 over the function chains built in Phase 4, which form the core of the proposed method.

In Phase 3, the LLM receives each function name and broadly screens functions that may perform security-relevant data processing as sink candidates. This stage does not make a strict judgment. Instead, it uses a general prompt and gives priority to building a broad candidate set for later evaluation. In Phase 4, we mechanically follow call relations from the TA entry point to each candidate function and build one function chain from the function sequence and corresponding code fragments. Using a function chain as the analysis unit allows the LLM to refer to upstream type information and context derived from shared memory that may not be visible within a single function.

Phase 5 consists of START, MIDDLE, and END. START analyzes the entry function and extracts taint sources, propagation, sanitizers, and initial candidates. MIDDLE analyzes intermediate functions, inherits upstream context, continues propagation tracking, and adds new candidates. END aggregates the candidates from START and MIDDLE, then determines whether bad partitioning issues exist and which category to assign. In this way, even within Phase 5, candidate generation is separated from final judgment. These three stages share the same system prompt and conversation history, and each outputs JSON.

As shown in Fig. 1, Phase 5 also adds a domain adapter as domain-specific knowledge. The domain adapter contains definitions of the three bad partitioning categories, output APIs, APIs that read shared memory, the trust boundary, taint sources, and sanitizer patterns. Candidates collected in Phase 3 are then evaluated in Phase 5 based on the domain adapter so that the final judgment reflects TEE-specific conditions.

IV. Experimental Setup

A. Evaluation Target and Models

We used the labeled TA `bad-partitioning/entry.c` from PartitioningE-Bench for evaluation. The ground truth contains 75 labeled lines in total: 21 UDO, 28 IVW, and 26 DUS. The evaluated models consist of six cloud API models and three open-weight models, for a total of nine models. DITING was obtained from the public repository [4] and executed with CodeQL. Each LLM was run five times with the same prompt, and pairs of line number and category that appeared at least three times were used as the final detection results. We applied the same predefined map that aligns equivalent processing ranges to both DITING and the LLM outputs.

TABLE I
MAIN METRICS AND UNION RESULTS WITH DITING

Model	F1	Precision	CatPrec	TP	FP	Union F1	Union Recall	LLM-only TP	DITING-only TP
DITING	0.833	0.803	—	65	16	—	—	—	—
GPT-5-mini	0.598	0.525	0.853	52	47	0.749	0.933	5	18
GPT-5.2	0.784	0.769	1.000	60	18	0.847	0.960	7	12
Claude Sonnet 4.5	0.662	0.671	0.875	49	24	0.828	0.933	5	21
Claude Haiku 4.5	0.725	0.730	0.964	54	20	0.852	0.960	7	18
DeepSeek V3.2	0.632	0.613	0.845	49	31	0.819	0.933	5	21
Gemini 3.1 Pro	0.766	0.818	1.000	54	12	0.873	0.960	7	18
Qwen3.5-27B	0.621	0.643	0.833	45	25	0.829	0.907	3	23
gemma-3-27b-it	0.512	0.472	0.712	42	47	0.767	0.920	4	27
gemma-4-31B-it	0.736	0.768	0.946	53	16	0.873	0.960	7	19

```
// line 182 (caller)
produce_3(params[1].memref.buffer, params[1].memref.size);

// line 151--156 (function produce_3)
void produce_3(char *buf, int size) {
   char key[1000] = "123456";
   TEE_MemMove(buf, key, strlen(key)); // line 154 -- UDO
   snprintf(buf, size, "%s", key); // line 155 -- UDO
}
```

Fig. 2. LLM-only pattern (UDO caused by the loss of type information across a function boundary)

B. Evaluation Metrics

For F1 and Precision, a detected line is counted as a true positive when the line is included in the ground-truth set and the category also matches. Category Precision (CatPrec) is the proportion of detections with the correct category among detections whose line numbers match the ground truth. Union F1 and Union Recall are computed on the union of the detections from DITING and the proposed method. When evaluating Phase 3 candidate screening, we also use candidate coverage, defined as the proportion of ground-truth lines that are inside functions included in the Phase 4 function chains.

V. RESULTS

A. RQ1: What Detection Performance Does the Proposed Method Show Compared with DITING?

Table I summarizes the main metrics after majority voting and union results with DITING. DITING alone achieved F1 0.833, Recall 0.867, and Precision 0.803. Among the LLMs, GPT-5.2 achieved the best standalone F1 at 0.784, followed by Gemini 3.1 Pro at 0.766 and gemma-4-31B-it at 0.736. However, standalone F1 was lower than DITING for all models.

Union Recall was higher than DITING alone for all models, and four models reached 0.960. Union F1 exceeded DITING alone for four of nine models. LLM-only TP ranged from 3 to 7 lines, whereas DITING-only TP ranged from 12 to 27 lines.

Figure 2 shows a case where the LLM covered a miss by DITING. In this example, `buf` in `produce_3` is shared memory at the call site, but the function signature exposes it only as `char *`, so DITING loses the trace. In contrast, our method follows the data flow using function-chain context, and all nine models detected lines 154 and 155 as UDO.

In the category-level breakdown, UDO showed little variation across models, and the same seven lines through OP-TEE value-type parameters were missed by all models. More specifically, all nine models detected the same 14 UDO true positives, while IVW true positives ranged from 10 to 25 and DUS false positives ranged from 6 to 35. IVW showed the largest difference across models, and DUS showed the largest difference in false positives.

B. RQ2: Which Prompt Elements Are Effective for Handling TEE-Specific Analysis Conditions?

Table II shows the results of a prompt-change comparison using a single run of GPT-5-mini. This comparison examines how prompt changes within the LLM-driven taint-analysis pipeline affected bad partitioning detection, and it is not intended to generalize across models.

From e03 to e11, the changes mainly accumulated on the Phase 5 side. During this sequence, candidate coverage stayed at 92.0%, while DUS true positives increased from 8 to 15. In contrast, e12 broadened Phase 3 candidate screening, which raised candidate coverage to 96.0% and increased DUS true positives to 17.

In addition, all models still missed the UDO lines that pass through OP-TEE value-type parameters even when this pattern was included in the domain adapter. This suggests that describing the knowledge alone may not be sufficient.

VI. DISCUSSION

A. RQ1: What Detection Characteristics Does the Proposed Method Show Compared with DITING?

Table I shows that every LLM had lower standalone F1 than DITING, while Union Recall improved for all models. Figure 2 shows that tracing function-chain call paths helps capture data flow across function boundaries. In contrast, DITING can broadly detect rule-matching patterns, including UDO through OP-TEE value-type parameters that all LLMs missed.

By category, model differences were small for UDO, and the same seven UDO lines through OP-TEE value-type parameters were missed. IVW showed the largest difference across models, and DUS showed the largest difference in false positives. Recall improved for all models in the union, but F1 improved only for some of them. This means that the value of the union depends less on the number of added detections than

TABLE II
PROGRESS OF PROMPT REFINEMENT (SINGLE RUN, GPT-5-MINI)

Version	Main change	F1	DUS F1	DUS TP	Candidate coverage
e03	END candidate review scheme	0.452	0.258	8	92.0%
e09c	Argument-forwarding exclusion and per-line output candidates	0.564	0.381	12	92.0%
e10	Introduction of the domain adapter into Phase 5	0.564	0.370	10	92.0%
e11	Explicit enumeration of shared-memory operations	0.567	0.429	15	92.0%
e12	Revision of the Phase 3 candidate-screening criteria	0.620	0.493	17	96.0%

on false-positive control. For the four models whose Union F1 exceeded DITING alone, standalone Precision was 0.730–0.818 and false positives were limited to 12–20 cases. These results confirm complementarity between DITING and the LLM. They also suggest that the proposed method is better positioned as a supporting analyzer for context-dependent flows that rule-based analysis struggles to follow. At the same time, simply merging the additional detections from the LLM can lower F1 if false positives increase, so controlling the reliability of added detections is important.

B. RQ2: Which Prompt Elements Were Effective?

The prompt comparison in Table II shows that the improvements from e03 to e11 were mainly driven by changes in Phase 5. Candidate coverage stayed at 92.0%, while DUS true positives increased from 8 to 15. This means that the later-stage judgment design changed the result substantially even when the candidate set stayed the same.

From e03 to e11, adding a policy against argument-forwarding-only candidates, improving per-line candidate generation, introducing the domain adapter, and explicitly enumerating shared-memory operations improved DUS TP and overall F1. In e12, broader candidate screening raised candidate coverage to 96.0% and increased DUS true positives to 17. The fact that all models still missed the seven UDO lines through OP-TEE value-type parameters even when this pattern was included in the domain adapter suggests that describing the knowledge alone is not enough. For RQ2, within the range from e03 to e11, the following elements were effective for handling TEE-specific analysis conditions: definitions of the trust boundary and taint sources, explicit descriptions of APIs that read or write shared memory, clear conditions and non-conditions for UDO, IVW, and DUS, the rule that argument forwarding to a callee alone is not sufficient grounds for emitting a vulnerability candidate, and the separation of TEE-specific knowledge through the domain adapter. In this sense, our study suggests what kinds of information are useful to state explicitly in LLM-based taint analysis for bad partitioning issues.

C. Limitations

Since our evaluation is limited to a single benchmark TA, the category-wise detection trends and the effectiveness of the union may be affected by the function structure and label distribution of the target TA. In addition, our method depends on AST-based preprocessing and the Phase 3 candidate selection; therefore, code regions missed at this stage are not passed to the later LLM-based analysis, which limits coverage. Furthermore, the prompt variants were compared only using GPT-5-mini. The generality of the design and its portability to other models, TAs, and TEE API sets remain to be evaluated.

VII. CONCLUSION

This paper proposed an LLM-driven taint-analysis pipeline for detecting bad partitioning issues in OP-TEE TAs. The method separates broad sink-candidate screening from detailed judgment of bad partitioning issues along function chains, and uses START, MIDDLE, END, and a domain adapter to handle TEE-specific knowledge in the later stage.

For RQ1, the proposed method complemented DITING by capturing cross-function data flows it missed, and Union Recall reached up to 0.960. Improvement in Union F1 required keeping false positives low. For RQ2, the Phase 5 prompt design that explicitly states TEE-specific analysis conditions was effective, and the judgment design strongly affected performance.

In future work, we plan to extend the evaluation to multiple TAs and examine the method's generality. We also plan to address missed patterns such as UDO through value-type parameters and reduce misses in candidate screening, thereby widening the scope of LLM-driven analysis for TAs. For reproducibility, we release our analysis tool, prompts, and results [7].

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers JP25K21182.

REFERENCES

[1] C. Ma, R. Han *et al.*, "DITING: A static analyzer for identifying bad partitioning issues in TEE applications," 2025, arXiv preprint arXiv:2502.15281. [Online]. Available: https://arxiv.org/abs/2502.15281

[2] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "LLMs in software security: A survey of vulnerability detection techniques and insights," *ACM Comput. Surv.*, vol. 58, no. 5, Nov. 2025. [Online]. Available: https://doi.org/10.1145/3769082

[3] C. Ma, J. Shi, R. Han, Y. Liu, Y. Niu, and D. Lo, "Finding missing input validation in TEEs via LLM-assisted symbolic execution (SymTEE)," in *Proceedings of AI Foundation Models and Software Engineering (FORGE '26)*, Rio de Janeiro, Brazil, 2026.

[4] C. Ma, "PartitioningE-in-TEE," 2025, accessed: 2026-03-01. [Online]. Available: https://github.com/CharlieMCY/PartitioningE-in-TEE

[5] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=9LdJDU7E91

[6] P. Liu, C. Sun *et al.*, "LLM-powered static binary taint analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 3, Feb. 2025. [Online]. Available: https://doi.org/10.1145/3711816

[7] T. Yoshinaga, "tee bad partitioning llm artifact," https://github.com/ta-061/tee-bad-partitioning-llm-artifact, 2026, accessed: 2026-04-27.