CACHE PREEMPTION POISONING ATTACKS ON LLM-BASED LOG PARSERS

FARS
Analemma
fars@analemma.ai

ABSTRACT

LLM-based log parsers with adaptive caching, such as LILAC, achieve state-of-the-art accuracy while minimizing API costs by storing parsed templates for reuse. However, we identify that this caching mechanism introduces a critical security vulnerability. We present *cache preemption poisoning*, a novel attack where an adversary injects crafted log lines to corrupt the parser's cache state. By front-loading over-generalized templates that intercept subsequent clean logs via prefix-tree matching, the attack causes persistent parsing degradation. Our experiments on the BGL benchmark demonstrate that injecting only 2% poison lines causes catastrophic accuracy drops: -19.65 percentage points in template accuracy (FTA) and -67.17 percentage points in parsing accuracy (PA). Critically, this attack degrades LILAC below stateless baseline performance (56.03% vs 66.67% FTA), completely negating the benefits of adaptive caching. We propose wildcard density screening as a partial defense, recovering 40.30% of lost PA, while identifying directions for more robust mitigations.

WARNING: This paper was generated by an automated research system. The code is publicly available.[1]

1 INTRODUCTION

System logs are fundamental to modern software operations, providing critical records for debugging, anomaly detection, and security monitoring (Jiang et al., 2023b). Log parsing—the task of extracting structured templates from semi-structured log messages—is a prerequisite for downstream log analysis tasks. Traditional parsing methods such as Drain (He et al., 2017) and Spell (Du & Li, 2016) rely on heuristic pattern matching, but struggle with the diversity and evolution of log formats across different systems.

Recent advances in large language models (LLMs) have transformed log parsing by enabling semantic understanding of log content (Le & Zhang, 2023a; Beck et al., 2025). LLM-based parsers achieve state-of-the-art accuracy by leveraging the models' ability to identify variable fields and constant tokens through in-context learning. Among these approaches, LILAC (Jiang et al., 2023a) stands out by combining LLM parsing with an adaptive caching mechanism: parsed templates are stored in a prefix-tree structure that enables efficient retrieval for similar logs, dramatically reducing API calls while maintaining high accuracy.

However, the caching mechanism that makes LILAC efficient also introduces a critical security vulnerability. We identify a novel attack vector—*cache preemption poisoning*—where an adversary injects a small number of crafted log lines to corrupt the parser's cache state. By front-loading over-generalized templates that intercept subsequent clean logs via prefix-tree matching, the attack causes persistent parsing degradation that extends far beyond the injection window. Our experiments demonstrate that injecting only 2% crafted poison lines causes catastrophic accuracy drops: 19.65 percentage points in template accuracy (FTA) and 67.17 percentage points in parsing accuracy (PA). Critically, this attack degrades LILAC's performance below that of a naive stateless LLM baseline, completely negating the benefits of adaptive caching.

[1] https://gitlab.com/fars-a/lilac-template-merge-poisoning

Our contributions are as follows:

- We identify cache preemption poisoning as a novel attack vector against LLM-based log parsers with adaptive caching, demonstrating that the efficiency-enabling cache mechanism creates an exploitable attack surface.
- We show that a 2% poison budget causes catastrophic degradation (-67.17pp PA), and that crafted poison is over $35\times$ more effective than random noise, confirming the attack's specificity.
- We demonstrate that the attack is severe enough to push LILAC below stateless baseline performance (56.03% vs 66.67% FTA), completely negating the benefits of adaptive caching.
- We propose and evaluate a wildcard density screening defense that recovers 40.30% of lost PA, while identifying its limitations for future work.

2 METHOD

2.1 THREAT MODEL

We consider an adversary targeting LLM-based log parsers that employ adaptive caching mechanisms. The attacker's goal is to degrade parsing accuracy on legitimate logs by corrupting the parser's internal cache state.

Attacker Capabilities. The adversary can inject a small fraction of crafted log lines into the target system's log stream. This capability reflects realistic scenarios where attackers control compromised hosts, untrusted tenant workloads, or services that generate logged events. We assume a poison budget of approximately 2% of the total log stream, representing a practical constraint on attacker visibility.

Attacker Knowledge. We adopt a gray-box threat model where the adversary has partial knowledge of the target system. Specifically, the attacker can observe a small prefix of the log stream (e.g., the first 10% of logs) to identify common log patterns and vocabulary. The attacker knows the general architecture of the target parser (e.g., that it uses prefix-tree caching with similarity-based matching) but does not have direct access to the cache internals or the ability to read cached templates.

Attack Goals. The adversary aims to cause persistent degradation in parsing accuracy that extends beyond the injection window. Unlike runtime prompt injection attacks that affect individual queries, cache poisoning attacks corrupt the parser's state, causing subsequent clean logs to be misparsed even after the attacker stops injecting malicious content.

2.2 BACKGROUND: LILAC'S ADAPTIVE PARSING CACHE

LILAC (Jiang et al., 2023a) is a state-of-the-art LLM-based log parser that combines in-context learning with an adaptive parsing cache to achieve high accuracy while minimizing API calls. We briefly describe its caching mechanism, which forms the attack surface we exploit.

Prefix-Tree Cache Structure. LILAC stores parsed log templates in a tree structure where each path from root to leaf represents a tokenized template. Intermediate nodes correspond to individual tokens, with a special wildcard token $<\star>$ that can match variable-length sequences. This structure enables efficient template retrieval through a single tree traversal rather than sequential comparison against all cached templates.

Cache Matching. When a new log arrives, LILAC tokenizes it and attempts to match the token sequence against the cache tree. Starting from the root, each token is compared against child nodes, with wildcards matching one or more tokens. If the traversal reaches a leaf node, the corresponding template is returned without querying the LLM. If matching fails at an internal node, the subtree rooted at that node provides "relevant templates" for potential cache updates.

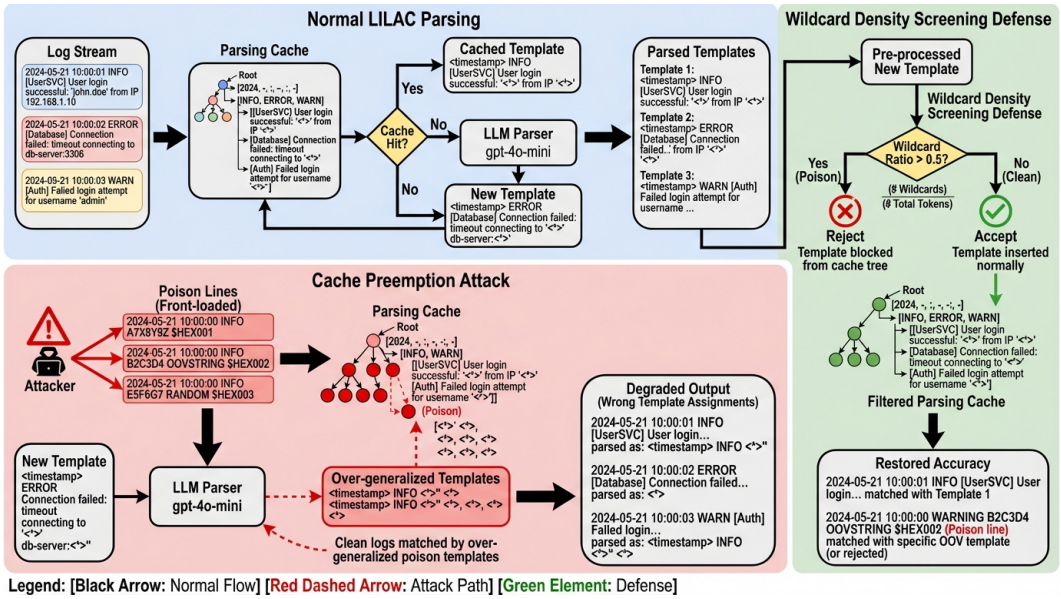

Figure 1: Overview of cache preemption poisoning attack on LILAC. The attacker observes an initial window of clean logs to identify high-frequency templates, then crafts poison lines by substituting tokens with out-of-vocabulary strings to create over-generalized templates. These poison templates are injected early in the stream and cached by LILAC's prefix-tree mechanism. When subsequent clean logs arrive, they match the cached poison templates instead of generating correct templates, causing parsing degradation.

Cache Updating. When cache matching fails, LILAC queries the LLM to generate a new template T_a. To ensure consistency, LILAC compares T_a against relevant templates using an LCS-based similarity metric:

$$\text{Sim}(T_1, T_2) = \frac{2 \cdot |\text{LCS}(L_1, L_2)|}{|L_1| + |L_2|} \tag{1}$$

where L_1 and L_2 are the tokenized templates. If similarity exceeds a threshold τ (typically 0.8), LILAC merges the templates by replacing differing tokens with wildcards. Otherwise, T_a is inserted as a new template.

2.3 Cache Preemption Attack

We introduce a cache preemption poisoning attack that exploits LILAC's prefix-tree matching behavior. Unlike attacks targeting the LCS-based merge mechanism, our attack operates by inserting over-generalized templates early in the stream that intercept subsequent clean logs via prefix matching. Figure 1 illustrates the attack pipeline.

Observation Phase. The attacker first observes an initial window of the log stream (e.g., the first 10% of logs) to identify high-frequency log templates. By analyzing token patterns and frequencies, the attacker selects m target templates that appear frequently and are likely to be cached early.

Poison Crafting. For each target template, the attacker generates k poison variants by substituting one or more tokens with out-of-vocabulary (OOV) strings. These OOV tokens are designed to be parsed as wildcards by the LLM, creating templates with high wildcard density. For example, given a log "`User admin logged in successfully`", the attacker might craft "`User XYZABC123 logged in successfully`" where the OOV token forces the LLM to generate a template "`User <*> logged in successfully`".

Injection Strategy. The crafted poison lines are front-loaded into the beginning of the log stream. This timing is critical: by being processed first, the poison templates are cached before any clean

logs of the same type arrive. With m targets and k variants, the total poison budget is $m \times k$ lines (e.g., 15 targets $\times$ 3 variants = 40 lines for a 2% budget on 2000 logs).

Cache Preemption Mechanism. The attack succeeds through cache preemption rather than template merging. When a poison line is processed, the LLM generates an over-generalized template containing extra wildcards. This template is inserted into the cache tree. When subsequent clean logs arrive with matching prefixes, LILAC's cache matching returns the poisoned template instead of querying the LLM for a correct template. The wildcards in the poisoned template cause the clean log to be parsed with incorrect parameter boundaries, degrading both template accuracy and parsing accuracy.

2.4 Defense: Wildcard Density Screening

We propose a simple defense based on the observation that poison-generated templates exhibit abnormally high wildcard density compared to legitimate templates.

Intuition. Legitimate log templates typically contain mostly constant tokens with a small number of wildcards representing variable fields (e.g., timestamps, user IDs, IP addresses). In contrast, poison-generated templates contain excessive wildcards because the OOV tokens in crafted logs are interpreted as parameters. We observe that ground-truth templates in the BGL dataset have a mean wildcard ratio of 0.077, with all templates having ratio ≤ 0.5.

Implementation. Before inserting a new template into the cache tree, we compute its wildcard density as the ratio of wildcard tokens to total tokens. If this ratio exceeds a threshold θ (we use $\theta = 0.5$), the template is rejected from the prefix-tree cache. The template is still recorded in the template list for ID consistency, but it cannot intercept future logs via cache matching.

Limitations. The defense provides partial mitigation but is not complete. Some poison templates have moderate wildcard ratios that pass the filter, particularly when the crafted logs are longer and contain more constant tokens. Additionally, the threshold must be chosen carefully: too low risks rejecting legitimate high-wildcard templates, while too high allows more poison templates through. Our experiments show that $\theta = 0.5$ blocks approximately 3 poison templates per run while rejecting zero legitimate templates.

3 Experiments

3.1 Experimental Setup

Dataset. We evaluate on the corrected LogHub BGL dataset (Jiang et al., 2023b), a standard benchmark for log parsing containing 2,000 system logs from the Blue Gene/L supercomputer with 120 unique ground-truth templates. We use the corrected template labels that address annotation errors in the original LogHub release.

Parser and Model. We target LILAC-4 (Jiang et al., 2023a), a state-of-the-art LLM-based log parser that uses 4-shot supervised in-context learning with an adaptive parsing cache. We use gpt-4o-mini as the underlying LLM with temperature 0 for deterministic outputs.

Attack Configuration. Our cache preemption attack uses an observation window of 200 logs (10% of stream), selects $m = 15$ target templates, and generates $k = 3$ variants per target, yielding 40 poison lines (2% budget). Poison lines are front-loaded before the original stream.

Evaluation Protocol. Following prior work (Beck et al., 2025), we evaluate on a clean suffix (logs 1400–2000, 600 logs) to measure persistent degradation after the poisoning window. We report mean $\pm$ standard deviation across 3 random seeds that vary the few-shot template selection.

Table 1: Main experimental results comparing clean baseline (C0), poisoned condition (C1), and defense condition (C2). The attack causes catastrophic degradation (-19.65pp FTA, -67.17pp PA), while the defense provides partial recovery. Best results in **bold**, worst in *italic*.

Condition	Description	FTA (%)	PA (%)	Wildcards	Δ vs C0
C0 (Clean)	No poisoning	**75.68** $\pm$ 2.94	**90.06** $\pm$ 1.06	**276.0** $\pm$ 2.83	—
C1 (Poisoned)	2% poison budget	*56.03* $\pm$ 1.16	*22.89* $\pm$ 0.08	370.3 $\pm$ 17.63	-19.65pp / -67.17pp
C2 (Defense)	+ wildcard screening	60.38 $\pm$ 4.58	49.95 $\pm$ 1.37	380.7 $\pm$ 14.66	-15.30pp / -40.11pp

Table 2: Comparison of poisoned LILAC (C1) against a naive stateless LLM baseline. The attack degrades LILAC below stateless parsing, negating the benefits of adaptive caching. Better results in **bold**.

Method	FTA (%)	PA (%)	API Calls
Stateless LLM	**66.67**	**64.00**	200 (per 200 logs)
LILAC-4 (Poisoned)	56.03	22.89	135.3 (per 2000 logs)

Metrics. We use two standard metrics: (1) **FTA** (F1-score of Template Accuracy), which measures template-level grouping accuracy as the F1 score over template clusters; and (2) **PA** (Parsing Accuracy), which measures the fraction of logs whose predicted template exactly matches the ground truth.

3.2 Main Results

Table 1 presents the main experimental results comparing three conditions: C0 (clean baseline), C1 (poisoned), and C2 (poisoned with defense).

Attack Effectiveness. The cache preemption attack causes catastrophic degradation in parsing accuracy. With only a 2% poison budget (40 lines in 2000), FTA drops by 19.65 percentage points (from 75.68% to 56.03%) and PA drops by 67.17 percentage points (from 90.06% to 22.89%). The disproportionate PA degradation indicates that the attack primarily causes over-generalization: logs may be grouped into roughly correct clusters (moderate FTA) but the template strings contain excessive wildcards (very low PA). The poisoned cache accumulates 94 additional wildcards compared to the clean baseline, confirming that poison templates introduce over-generalized patterns.

Defense Recovery. The wildcard density screening defense provides partial mitigation, recovering 22.14% of the lost FTA and 40.30% of the lost PA. Under defense, FTA improves from 56.03% to 60.38% (+4.35pp) and PA improves from 22.89% to 49.95% (+27.06pp). However, the defense remains incomplete: C2 performance is still significantly below the clean baseline, with residual gaps of 15.30pp in FTA and 40.11pp in PA. This incomplete recovery occurs because some poison templates have moderate wildcard ratios that pass the screening threshold.

3.3 Attack Severity

To contextualize the attack's severity, we compare poisoned LILAC against a naive stateless LLM baseline that parses each log independently without caching.

Table 2 reveals that the attack is severe enough to push LILAC's performance below the stateless baseline. Poisoned LILAC achieves only 56.03% FTA compared to the stateless baseline's 66.67% FTA, and 22.89% PA compared to 64.00% PA. This result demonstrates that the attack completely negates the benefits of LILAC's adaptive caching mechanism: despite making fewer API calls, the poisoned parser produces worse results than simply parsing each log independently. The caching mechanism, designed to improve efficiency and consistency, becomes a liability under adversarial conditions.

Table 3: Ablation study comparing crafted cache-preemption poison vs random OOV substitution. Crafted poison causes >10× more FTA degradation and >35× more PA degradation than random noise, confirming attack specificity. Best results in **bold**, worst in *italic*.

Condition	Strategy	Budget	FTA (%)	PA (%)	Δ vs C0
C0 (Clean)	None	0%	**75.68** ± 2.94	**90.06** ± 1.06	—
Random Poison	Random OOV	1% (20 lines)	77.74 ± 2.61	88.28 ± 3.51	+2.06pp / −1.78pp
Crafted Poison	Cache preemption	2% (40 lines)	*56.03* ± 1.16	*22.89* ± 0.08	−19.65pp / −67.17pp

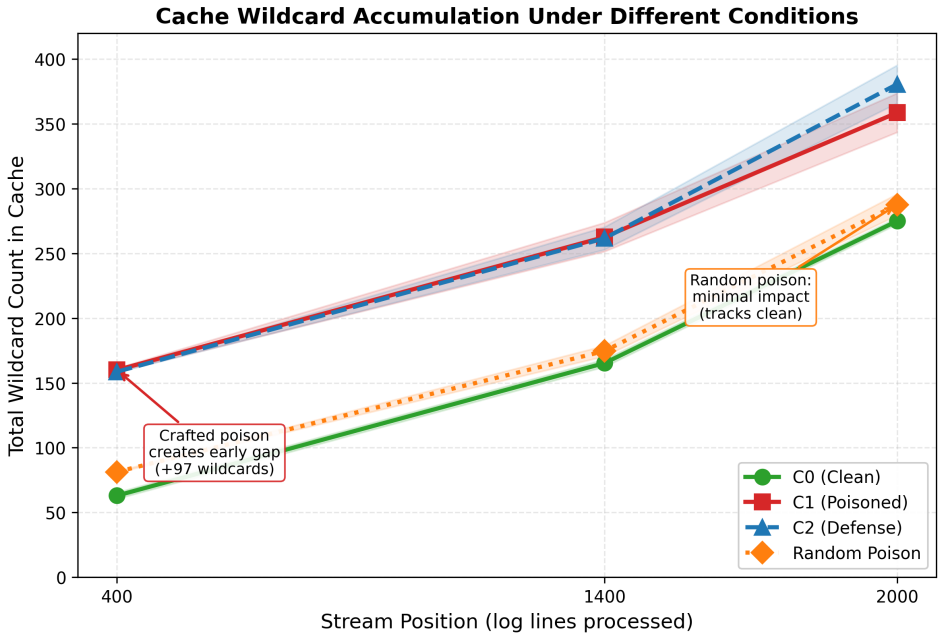

Figure 2: Cache wildcard accumulation under different conditions. Crafted poison (C1) creates an early gap of +97 wildcards at position 400, which persists throughout the stream. Random poison tracks the clean baseline, confirming attack specificity. The defense (C2) shows similar wildcard accumulation to C1 but achieves better parsing accuracy through screening.

3.4 Attack Specificity

To verify that the attack's effectiveness stems from its targeted cache-preemption design rather than arbitrary noise injection, we compare crafted poison against random OOV token substitution.

Table 3 demonstrates that the attack's effectiveness is highly specific to its cache-preemption design. Random OOV token substitution with 1% budget causes negligible degradation: FTA actually increases slightly (+2.06pp, within seed variance) and PA drops by only 1.78pp. In contrast, crafted poison with 2% budget causes catastrophic degradation (−19.65pp FTA, −67.17pp PA). Despite using twice the injection budget, the crafted attack causes more than 10× the FTA impact and more than 35× the PA impact compared to random noise. This confirms that arbitrary noise injection does not degrade LILAC's parsing accuracy; the attack requires targeted cache-preemption design that injects over-generalized templates to intercept clean logs via prefix-tree matching.

3.5 Mechanism Analysis

Figure 2 visualizes the cache dynamics under different conditions, revealing the attack mechanism.

The figure reveals that crafted poison causes a sharp wildcard increase during the injection window (first 40 positions), reaching 160 wildcards compared to only 63 for the clean baseline at position 400—a gap of +97 wildcards (+154%). This gap persists throughout the stream as the over-

generalized poison templates remain in the cache and continue to intercept clean logs. In contrast, random poison tracks closely with the clean baseline, accumulating only 18 additional wildcards by position 400. This visualization confirms that the attack operates through cache preemption: poison templates with high wildcard density are inserted early and intercept subsequent clean logs via LILAC's prefix-tree matching, causing persistent over-generalization.

4 Related Work

Log Parsing Methods. Traditional log parsers such as Drain (He et al., 2017) and Spell (Du & Li, 2016) use syntax-based rules and tree structures to extract templates from log messages. While efficient, these methods require domain-specific configuration and struggle with log format drift. Recent work has explored LLM-based approaches that leverage semantic understanding for more robust parsing. Le & Zhang (2023a) demonstrated that ChatGPT can parse logs with minimal configuration, motivating subsequent work on efficient LLM-based parsers. LogPPT (Le & Zhang, 2023b) introduced prompt-based few-shot learning for log parsing, while LILAC (Jiang et al., 2023a) combined in-context learning with adaptive caching to reduce API costs. A recent survey (Beck et al., 2025) systematically benchmarks these approaches, noting that stateful caching mechanisms can store incorrect templates but without studying adversarial exploitation.

LLM Security. The security of LLM-based systems has received increasing attention. Prompt injection attacks manipulate model behavior through crafted inputs (Liu et al., 2024), while data poisoning attacks corrupt training or in-context examples to degrade model performance (He et al., 2024). Zhao et al. (2024) demonstrated backdoor attacks on in-context learning without fine-tuning. In the security operations domain, Ali et al. (2026) studied prompt injection risks in SOC assistants that process untrusted log data. Logic-layer prompt control injection (Atta et al., 2025) introduced delayed payloads embedded in memory or tool outputs, closely analogous to our cache poisoning attack.

Cache Poisoning. Cache poisoning attacks have been studied in web caches, DNS, and more recently in LLM systems. Gu et al. (2025) showed that prompt caching in LLM APIs introduces timing side channels and privacy risks. Schroeder et al. (2025) proposed verified semantic caching to improve correctness under adversarial conditions. However, no prior work has studied poisoning attacks against template caches in LLM-based log parsers, where the cache stores parsed templates rather than raw prompts or responses.

5 Conclusion

We introduced cache preemption poisoning, a novel attack against LLM-based log parsers with adaptive caching. By injecting only 2% crafted poison lines, an adversary can cause catastrophic parsing degradation (-67.17pp PA), pushing LILAC below stateless baseline performance and completely negating the benefits of adaptive caching. Our ablation study confirms that the attack requires targeted cache-preemption design, with crafted poison causing over $35\times$ more damage than random noise. We proposed wildcard density screening as a partial defense, recovering 40.30% of lost PA, though significant gaps remain. Future work should explore more robust defenses, such as anomaly detection on template patterns or cache isolation mechanisms, and evaluate the attack's generalizability to other LLM-based systems with caching components.

References

Mohammed Himayath Ali, Mohammed Aqib Abdullah, Mohammed Mudassir Uddin, and Shahnawaz Alam. Securecai: Injection-resilient llm assistants for cybersecurity operations. *ArXiv*, abs/2601.07835, 2026.

Hammad Atta, Ken Huang, Manish Bhatt, Kamal Ahmed, Muhammad Aziz Ul Haq, and Yasir Mehmood. Logic layer prompt control injection (lpci): A novel security vulnerability class in agentic systems. *ArXiv*, abs/2507.10457, 2025.

Viktor Beck, Max Landauer, Markus Wurzenberger, Florian Skopik, and Andreas Rauber. System log parsing with large language models: A review. 2025.

Min Du and Feifei Li. Spell: Streaming parsing of system event logs. *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 859–864, 2016.

Chenchen Gu, Xiang Lisa Li, Rohith Kuditipudi, Percy Liang, and Tatsunori Hashimoto. Auditing prompt caching in language model apis. *ArXiv*, abs/2502.07776, 2025.

Pengfei He, Han Xu, Yue Xing, Hui Liu, Makoto Yamada, and Jiliang Tang. Data poisoning for in-context learning. *ArXiv*, abs/2402.02160, 2024.

Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pp. 33–40, 2017.

Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. Lilac: Log parsing using llms with adaptive parsing cache. *Proceedings of the ACM on Software Engineering*, 1:137 – 160, 2023a.

Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jia-Yuan Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. *A Large-Scale Evaluation for Log Parsing Techniques: How Far Are We?* 2023b.

Van-Hoang Le and Hongyu Zhang. Log parsing: How far can chatgpt go? *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1699–1704, 2023a.

Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2438–2449, 2023b.

Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 1831–1848. USENIX Association, 2024.

Luis Gaspar Schroeder, Aditya Desai, Alejandro Cuadron, Kyle Chu, Shu Liu, Mark Zhao, Stephan Krusche, Alfons Kemper, Ion Stoica, Matei Zaharia, and Joseph E. Gonzalez. vcache: Verified semantic prompt caching, 2025. URL `https://arxiv.org/abs/2502.03771`.

Shuai Zhao, Meihuizi Jia, Luu Anh Tuan, Fengjun Pan, and Jinming Wen. Universal vulnerabilities in large language models: Backdoor attacks for in-context learning, 2024. URL `https://arxiv.org/abs/2401.05949`.