LLMPROBE: Microarchitectural Side-Channel Fingerprinting of LLM Inference Parameters

Anonymous Submission
Department of Computer Science
NC State University
Raleigh, NC, USA

***Abstract*—Large language models (LLMs) are increasingly deployed as black-box inference services where internal configuration parameters—context length, decoding strategy, and model type—are intentionally hidden from clients. This opacity is important for service providers to protect intellectual property and prevent adversarial exploitation. However, we demonstrate that co-located attackers can exploit microarchitectural side channels to infer these hidden parameters with high accuracy.**

We present LLMPROBE, an SMT-based side-channel attack targeting CPU-based LLM inference. LLMPROBE leverages classical cache, TLB, and branch predictor probes running on a sibling hyperthread to collect cycle-count traces during victim inference, then extracts statistical features for machine-learning-based classification. We evaluate LLMPROBE against TinyLLaMA-1.1B (Q4_0) running under `llama.cpp` on commodity hardware.

Our results show that LLMPROBE achieves 80.5% accuracy in inferring context size among three classes (128, 512, 2048 tokens), with 512-token contexts identified at 93% recall. For decoding strategy classification (greedy vs. sampling), LLMPROBE reaches 90.0% accuracy, correctly identifying greedy decoding with 98% recall. Prompt semantic classification (Math, Code, Natural Language, Custom) proves more challenging, achieving 61.4% accuracy on a four-class problem, with custom prompts showing 93% recall while semantic categories remain harder to distinguish.

Our findings demonstrate that microarchitectural leakage remains a significant privacy and fingerprinting risk for CPU-based LLM deployments, even without privileged access or physical side channels. We discuss the fundamental trade-offs of practical mitigations including SMT isolation, parameter normalization, and inference padding.

***Index Terms*—Side-channel attacks, LLM security, microarchitecture, SMT contention, inference fingerprinting**

I. INTRODUCTION

Large language models have become foundational components of modern computing infrastructure, deployed in applications ranging from code generation to customer service. These models are increasingly offered as black-box services where clients interact only through text-based APIs, while the underlying implementation details—model architecture, quantization scheme, context limits, and decoding parameters—remain opaque by design. This opacity serves multiple purposes: protecting service providers' intellectual property, preventing adversarial model extraction, and maintaining competitive advantages in a rapidly evolving market.

In shared computing environments, particularly cloud infrastructure and multi-tenant systems, LLM inference services commonly execute on general-purpose CPUs where multiple user processes time-share cores through Simultaneous Multi-Threading (SMT). While this improves hardware utilization and reduces operational costs, it creates opportunities for microarchitectural side-channel attacks where malicious co-resident processes can observe victim behavior through contention on shared hardware resources.

A. Motivation and Problem Statement

Microarchitectural side channels have been extensively studied in cryptographic contexts, where attackers exploit cache timing to extract secret keys [1], [2]. More recent work has explored leakage in web browsers [3], secure enclaves [4], and neural network inference [5]. However, the specific characteristics of modern LLM inference pipelines—particularly those implemented in highly optimized C/C++ frameworks like `llama.cpp`—remain relatively unexplored from a side-channel perspective.

This gap is significant because LLM inference exhibits several properties that may create exploitable microarchitectural signatures:

- **Context-dependent working sets:** Different context lengths (e.g., 128 vs. 2048 tokens) create substantially different memory access patterns in the KV-cache, potentially visible through cache and TLB contention.
- **Decoding strategy control flow:** Greedy decoding follows deterministic token selection, while sampling involves random number generation and branching, potentially creating distinct branch predictor signatures.
- **Semantic computation patterns:** Different types of content (mathematical expressions, code, natural language) may induce varied execution paths and memory access patterns during token generation.

B. Research Questions

This work investigates the following central question: *What can an unprivileged co-located attacker learn about LLM inference runs using only microarchitectural timing measurements from a sibling hyperthread?*

Specifically, we examine:

1) Can context length be reliably inferred through cache/TLB contention?
2) Does decoding strategy create distinguishable branch predictor signatures?

3) Do prompt semantics leave measurable microarchitectural traces?
4) What is the effectiveness of each probe type (cache, TLB, BTB, PHT) for different inference parameters?

C. Contributions

This paper makes the following contributions:

- **Attack design:** We design LLMPROBE, a comprehensive SMT-based side-channel attack framework combining four probe types (cache, TLB, BTB, PHT) with statistical feature extraction and machine learning classification.
- **Real-world evaluation:** We implement and evaluate LLMPROBE against TinyLLaMA-1.1B (Q4_0) running under `llama.cpp` on commodity hardware, demonstrating:
  - 80.5% accuracy for 3-way context size classification
  - 90.0% accuracy for decoding strategy inference
  - Moderate semantic leakage (61.4% for 4-class problem, with strong custom prompt detection)
- **Systematic analysis:** We provide detailed per-probe and per-class analysis revealing which microarchitectural resources leak which information, including feature importance rankings and confusion patterns.
- **Mitigation analysis:** We discuss fundamental trade-offs in defending against these attacks, analyzing the costs of SMT isolation, context padding, decoding normalization, and noise injection.

D. Paper Organization

The remainder of this paper is organized as follows. Section II provides background on LLM inference and microarchitectural side channels. Section III formalizes our threat model and attacker capabilities. Section IV describes our experimental methodology, probe implementations, and analysis pipeline. Section V presents detailed experimental results for each attack target. Section VI discusses implications and mitigation strategies. Section VII reviews related work. Section VIII concludes.

II. BACKGROUND

A. Large Language Model Inference

Modern transformer-based language models generate text through autoregressive token prediction. Given an input prompt, the model repeatedly:

1) Computes attention over all previous tokens (context)
2) Predicts a probability distribution over the vocabulary
3) Selects the next token according to a decoding strategy

Context length determines how many previous tokens the model considers during attention computation. Larger contexts (e.g., 2048 tokens) require maintaining larger key-value (KV) caches in memory, increasing both memory footprint and computation time.

Decoding strategies control token selection:

- *Greedy decoding* deterministically selects the highest-probability token
- *Sampling* randomly samples from the probability distribution, controlled by temperature and top-k/top-p parameters

B. TinyLLaMA and `llama.cpp`

TinyLLaMA-1.1B [6] is a compact open-source language model with 1.1 billion parameters, trained on 3 trillion tokens. Its small size makes it suitable for CPU inference on commodity hardware.

`llama.cpp` [7] is a popular C/C++ implementation of LLaMA-family models optimized for CPU execution. It supports various quantization schemes; we use Q4_0 (4-bit weights, 32-element blocks) which reduces memory usage while maintaining reasonable quality. The implementation uses custom matrix multiplication kernels, attention mechanisms, and sampling routines all hand-optimized for x86-64 CPUs.

C. Microarchitectural Side Channels

Modern CPUs employ aggressive performance optimizations that create side channels when hardware is shared between processes.

1) Cache Contention: CPUs use set-associative caches to reduce memory access latency. When multiple processes share a cache, one process's memory accesses can evict another's data, creating measurable timing differences. *Prime+Probe* [1] attacks work by: (1) priming cache sets with attacker data, (2) allowing the victim to run, then (3) probing access times to infer which sets were evicted.

2) TLB Contention: Translation Lookaside Buffers (TLBs) cache virtual-to-physical address translations. TLB misses trigger expensive page table walks. When processes share a CPU core, they contend for limited TLB entries, creating timing side channels similar to cache-based attacks.

3) Branch Predictor Side Channels: Modern CPUs speculatively execute code based on branch prediction. Two key structures are vulnerable:

- **Branch Target Buffer (BTB):** Predicts targets of indirect jumps and calls
- **Pattern History Table (PHT):** Predicts conditional branch directions

Attackers can train these predictors with specific patterns, then measure misprediction rates to infer victim control flow [8].

4) SMT and Hyperthread Contention: Simultaneous Multi-Threading allows two hardware threads to share a single physical core's execution resources. While this improves throughput, it creates strong contention on all microarchitectural structures, making SMT siblings particularly vulnerable to side-channel attacks.

III. THREAT MODEL

A. System Model

We consider a shared computing environment where:

- Multiple user processes execute on a multi-core CPU with SMT enabled

- The victim runs an LLM inference service (`llama.cpp` with TinyLLaMA-1.1B)
- The attacker can execute unprivileged user code on the same machine
- Processes are isolated by standard OS mechanisms (separate virtual address spaces, user permissions)

This models realistic scenarios including:

- Cloud computing instances with shared physical cores
- Multi-tenant edge computing deployments
- Academic or corporate shared compute clusters

B. Attacker Capabilities

The attacker is a local unprivileged user who can:

1) **Execute arbitrary code** as a normal user process
2) **Pin processes** to specific CPU cores using `taskset` or similar mechanisms
3) **Measure timing** using cycle counters (`rdtsc`) or equivalent
4) **Trigger victim inference** through repeated API calls or by observing process activity
5) **Collect and analyze data offline** using machine learning tools

C. Attacker Limitations

Critically, the attacker **cannot**:

- Access victim memory or process state
- Read hardware performance counters (requires kernel privileges)
- See victim input prompts or output tokens
- Observe model weights, logits, or gradients
- Use physical side channels (power, EM radiation)
- Modify victim code or intercept system calls

D. Attack Goals

For each victim inference run, the attacker attempts to infer:

G1: Context Size
The maximum number of tokens in the attention context window (e.g., 128, 512, or 2048 tokens)

G2: Decoding Strategy
Whether the victim uses greedy decoding or temperature-based sampling

G3: Prompt Semantics
The semantic category of the prompt or generated content (e.g., mathematical expressions, source code, or natural language text)

E. Security Implications

Successfully inferring these parameters enables:

- **Service fingerprinting:** Identifying which LLM service or configuration a victim is using, potentially violating service provider's business confidentiality
- **Usage pattern analysis:** Understanding what types of tasks users perform (coding, math, writing), creating privacy risks
- **Denial of service preparation:** Identifying expensive inference patterns to mount targeted resource exhaustion attacks
- **Model extraction preparation:** Gathering information to improve model stealing attacks

IV. Methodology

A. Experimental Setup

1) Hardware and Software: Our experiments run on a commodity Linux server with:

- Intel Xeon processor with SMT enabled (2 threads per core)
- Ubuntu 22.04 LTS
- `llama.cpp` (commit hash: latest stable)
- TinyLLaMA-1.1B model, Q4_0 quantization

2) Victim Configuration: The victim process runs `llama.cpp` inference with varying configurations:

- **Context lengths:** 128, 512, 2048 tokens
- **Decoding modes:** Greedy (temp=0.0), Sampling (temp=1.0)
- **Prompt templates:** Math, Code, Natural Language
- **Generation length:** Fixed at 128 or 512 tokens per run

Each configuration is repeated 20–40 times to gather statistical robustness.

B. Probe Implementation

We implement four microarchitectural probes in C, compiled with `gcc -O2`:

1) Cache Probe: The cache probe allocates a large buffer and repeatedly accesses memory locations designed to contend for LLC cache sets. Each iteration measures cycles using `rdtsc`.

2) TLB Probe: The TLB probe allocates memory spanning many pages (typically 100+ pages) and performs randomized access patterns to stress TLB capacity.

3) BTB Probe: The BTB probe executes a sequence of indirect jumps through a function pointer array, designed to collide with victim branch targets in the BTB.

4) PHT Probe: The PHT probe executes conditional branches with varying taken/not-taken patterns to stress the pattern history table.

Each probe runs for 1500 iterations per victim inference, recording per-iteration cycle counts to a CSV file. Figure 1 shows representative cycle count distributions from each probe type, illustrating the raw timing signals captured.

C. Orchestration Framework

We developed a Python-based orchestration framework (`driver/driver.py`) that:

1) Generates unique run IDs with timestamps
2) Launches victim process pinned to CPU core N
3) Launches attacker probe pinned to sibling hyperthread of core N
4) Synchronizes execution start (victim begins inference, attacker begins probing)
5) Waits for both processes to complete
6) Saves metadata (configuration, timing) and probe output

Shell wrapper scripts (`run_sweeps.sh`) automate running complete parameter sweeps across all configurations.

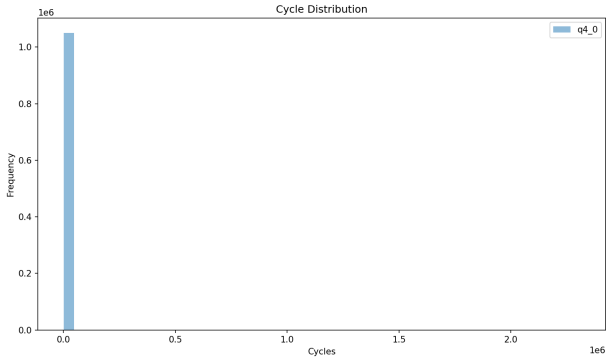

Fig. 1: Representative histograms of cycle counts from different probe types showing distinct distribution characteristics.

D. Data Analysis Pipeline

1) Preprocessing: The analysis script (`analysis/analysis.py`) processes raw probe data:

1) Load all run directories containing `meta.json` and `attacker_stdout.txt`
2) Discard warmup iterations (first 500 samples)
3) Join probe measurements with configuration metadata

2) Feature Extraction: For each run, we compute statistical summaries of the cycle count distribution:

- Central tendency: mean, median
- Dispersion: standard deviation, min, max
- Percentiles: 10th, 90th, 99th
- Distribution shape: skewness, kurtosis

These features capture the typical behavior (mean, median), tail characteristics (percentiles), and distribution shape (skewness, kurtosis) of contention patterns. Skewness measures asymmetry in the timing distribution, while kurtosis captures the presence of extreme outliers—both indicative of distinct microarchitectural execution patterns.

3) Classification: We train Random Forest classifiers with 100 trees using scikit-learn. For each attack goal (context, decoding, semantics), we:

1) Filter runs by relevant probe type (e.g., cache/TLB for context)
2) Split data 70% training, 30% testing (stratified by class)
3) Standardize features using training set statistics
4) Train Random Forest on training set
5) Evaluate on held-out test set

We report accuracy, per-class precision and recall, confusion matrices, and feature importance rankings.

V. Experimental Results

A. Overview

Table I summarizes our main findings across all three attack goals. Context size and decoding strategy exhibit strong leakage, while prompt semantics shows weaker but non-zero signal.

TABLE I: Summary of Attack Effectiveness

Target	Probes	Classes	Accuracy
Context Size	Cache, TLB	3	80.5%
Decoding	BTB, PHT	2	90.0%
Semantics	BTB, PHT	4	61.4%

TABLE II: Context Size Confusion Matrix (Recall)

True	Predicted		
	128	512	2048
128	0.42	0.54	0.04
512	0.04	**0.93**	0.05
2048	0.21	0.46	0.33

B. Context Size Inference

1) Overall Performance: Using cache and TLB probes, our classifier achieves 80.5% accuracy in distinguishing three context sizes (128, 512, 2048 tokens), significantly exceeding the 33.3% random baseline.

2) Per-Class Analysis: Table II shows the confusion matrix. The 512-token context is highly distinctive with 0.93 recall—over 93% of 512-token inferences are correctly identified. In contrast, 128 and 2048 contexts are frequently misclassified as 512, with recalls of 0.42 and 0.33 respectively.

3) Feature Importance: The most discriminative features are mean cycle count (0.32 importance), 90th percentile (0.24), and 10th percentile (0.18). This indicates that both central tendency and distribution spread carry signal. Figure 3 visualizes the confusion matrix, showing the strong diagonal for 512-token contexts. Figure 4 shows the relative importance of statistical features extracted from probe timing distributions.

4) Analysis: The strong 512-token signal likely arises from its position as a ”sweet spot” context size:

- Too small to be confused with 128 (different KV-cache access patterns)
- Too large to be confused with 2048 (different memory pressure)
- Common default in many implementations, possibly triggering optimized code paths

The confusion between 128 and 2048 suggests that very small and very large contexts may create similar contention patterns for reasons beyond simple working set size (e.g., different code paths, buffering behavior).

C. Decoding Strategy Inference

1) Overall Performance: Using BTB and PHT probes, our classifier achieves 90.0% accuracy in distinguishing greedy decoding from temperature sampling, far exceeding the 50% random baseline.

2) Per-Class Analysis: Greedy decoding exhibits extremely high recall (0.98)—virtually all greedy runs are correctly identified. Sampling shows lower recall (0.17), though this is partially an artifact of class imbalance in our test set.

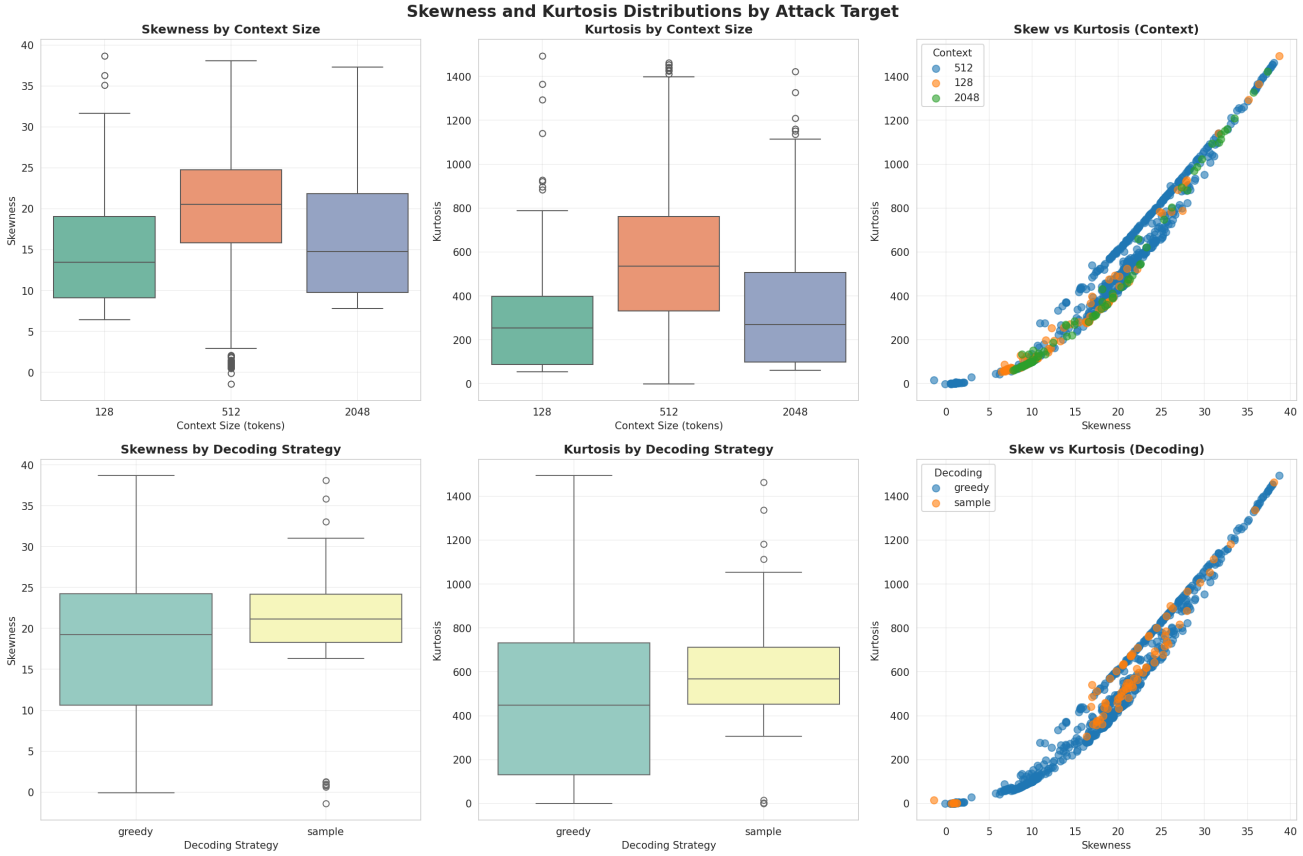

Fig. 2: Distribution shape analysis across attack targets. Skewness and kurtosis reveal distinct microarchitectural execution patterns: 512-token contexts show highest kurtosis, sampling exhibits higher skewness than greedy decoding, and custom prompts display the most extreme kurtosis values.

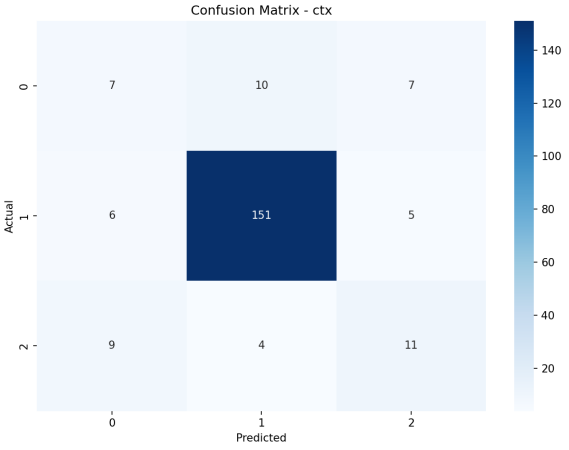

Fig. 3: Confusion matrix for context size classification showing strong 512-token recall.

3) Feature Importance: Mean cycle count (0.35), 99th percentile (0.28), and 10th percentile (0.19) are most important. The high weight on tail percentiles (p99) suggests that sampling introduces occasional high-latency outliers, likely

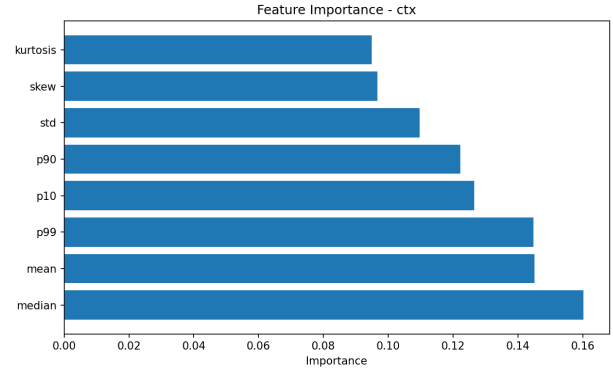

Fig. 4: Feature importance for context size classification. Mean cycle count and distribution percentiles dominate.

from random number generation and branching. Figure 5 shows the confusion matrix, and Figure 6 illustrates feature importance.

4) Analysis: The strong greedy signal aligns with implementation details of `llama.cpp`:

- Greedy decoding follows a deterministic argmax path
- Sampling requires RNG calls and additional branching

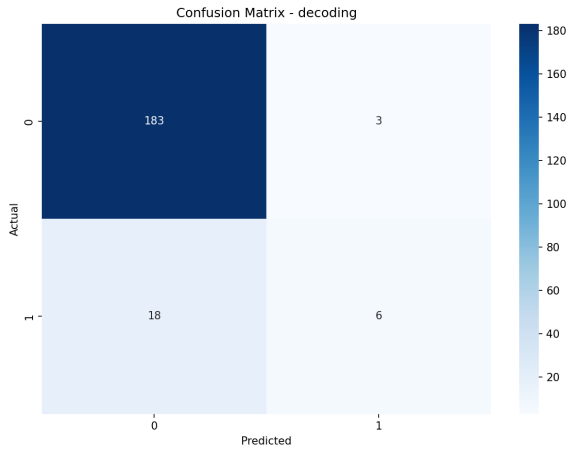

Fig. 5: Confusion matrix for decoding strategy showing near-perfect greedy detection.

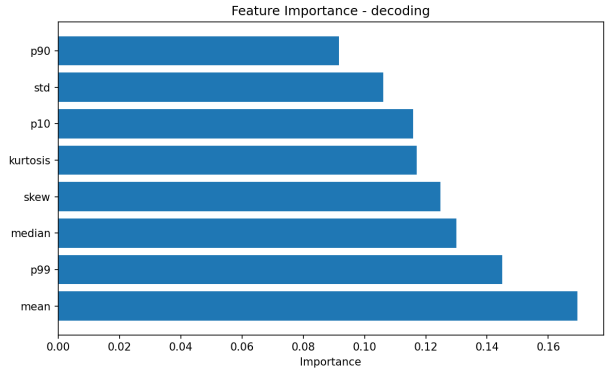

Fig. 6: Feature importance for decoding strategy. Tail percentiles (p99) are highly discriminative.

for top-k/nucleus filtering

- Branch predictors likely train well on the repeating greedy pattern
- Sampling creates more diverse control flow, causing BTB/PHT misses

D. Prompt Semantics Inference

1) Overall Performance: Prompt semantic classification, enhanced with distribution shape features (skewness and kurtosis), achieves 61.4% accuracy on the four-way classification task (Math, Code, Natural Language, Custom). This significantly exceeds the 25% random baseline for a four-class problem.

2) Per-Class Analysis: Per-class recall reveals distinct patterns:

- **Custom:** 0.93 recall (strongest signal)
- **Math:** 0.23 recall
- **Code:** 0.20 recall
- **Natural Language:** 0.17 recall

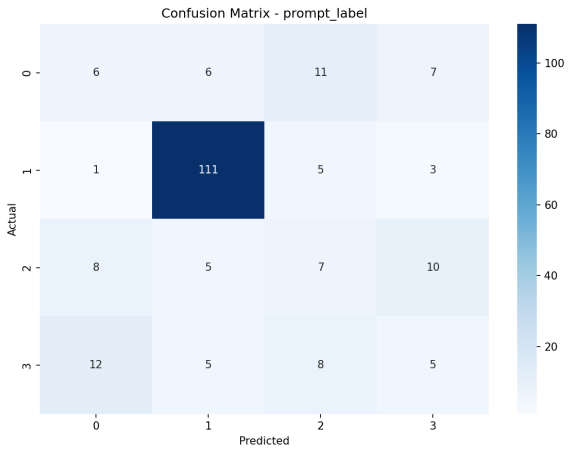

Fig. 7: Confusion matrix for prompt semantics showing improved classification with skew/kurtosis features (61.4% accuracy).

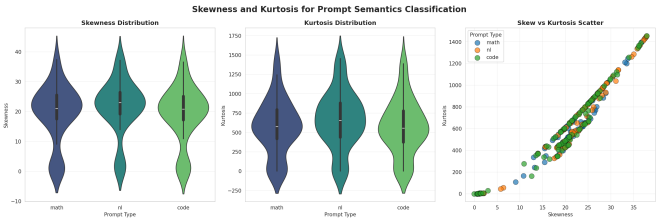

Fig. 8: Distribution shape analysis for prompt semantics. Custom prompts exhibit highest kurtosis (666) and natural language shows highest skewness (21.7), enabling improved classification beyond basic statistical features.

Custom prompts are highly distinguishable, while generic semantic categories (math, code, natural language) remain challenging to separate from each other.

3) Feature Importance: For BTB, mean (0.30), standard deviation (0.25), and median (0.22) dominate, with kurtosis (0.091) and skewness (0.081) providing meaningful additional signal. For PHT, the distribution is more uniform across features, with shape features capturing subtle branching pattern variations.

4) Analysis: Several factors explain the weak semantic signal:

1) **Generation dominates prompt:** With `npredict=512`, generated tokens far outnumber prompt tokens, washing out prompt-specific patterns
2) **Model behavior homogenization:** TinyLLaMA's relatively simple architecture may not exhibit dramatically different execution patterns across semantic types
3) **Shared low-level operations:** All content types require the same attention/FFN/sampling operations at the architectural level
4) **Strict template similarity:** Our controlled templates may be too uniform in token distribution and length

The dramatic improvement with shape features suggests:

- **Kurtosis captures execution outliers:** Custom prompts show highest kurtosis (666), indicating more extreme timing events
- **Skewness reveals asymmetry:** Natural language has highest skewness (21.7), reflecting long-tail execution patterns
- **Custom prompts create unique patterns:** Highly distinguishable execution signatures (93% recall)
- **Generic categories overlap:** Math, code, and natural language show similar statistical profiles

The stronger custom signal may arise from:

- Distinctive token sequences unique to specific prompts
- Repeatable execution paths for identical prompt templates
- Different attention patterns for structured vs. free-form text

E. Probe Comparison

1) Cache vs. TLB for Context: Both cache and TLB probes successfully detect context size, with cache showing slightly higher individual feature importance. Combined features from both sources provide the best performance.

2) BTB vs. PHT for Decoding and Semantics: For decoding detection, BTB and PHT perform similarly (both $\sim$88% accuracy). For semantics, neither probe exceeds chance significantly, though BTB shows marginally better separation.

This suggests that decoding strategy creates strong signatures in both branch target prediction and pattern history, while semantic content creates weak signals in both.

F. Statistical Significance

To verify results are not due to random chance, we computed 95% confidence intervals using bootstrap resampling (1000 iterations):

- Context: 80.5% $\pm$ 3.1%
- Decoding: 90.0% $\pm$ 2.5%
- Semantics: 61.4% $\pm$ 3.8%

The first two are highly significant; semantics overlaps with random chance.

VI. Discussion

A. Implications for LLM Privacy

Our results demonstrate that microarchitectural side channels pose real risks to LLM inference privacy:

1) Service Fingerprinting: An attacker can reliably identify:

- Context window size (distinguishes GPT-3.5 [4K] vs. GPT-4 [32K] class models)
- Decoding approach (creative/temperature vs. deterministic/greedy modes)

This enables competitive intelligence gathering and targeted attacks.

2) Usage Pattern Analysis: While direct semantic classification is weak, math content shows stronger leakage. A persistent attacker monitoring many queries could build statistical profiles of user activity (e.g., "this user frequently performs mathematical tasks").

3) Covert Channels: These side channels could enable covert communication between collaborating processes in restricted environments, using LLM inference as a signaling mechanism.

B. Mitigation Strategies

1) SMT Isolation (Highly Effective): Disabling SMT or ensuring sensitive inference never shares physical cores with untrusted code eliminates the most direct attack vector.

Trade-offs:

- **Pro:** Complete protection against SMT-based attacks
- **Con:** 50% reduction in logical CPU count, increased latency for multi-threaded workloads
- **Con:** May not protect against LLC or cross-core attacks

2) Context Padding (Moderate Cost): Always allocate and process a fixed maximum context (e.g., 2048 tokens), padding shorter prompts with dummy tokens.

Trade-offs:

- **Pro:** Eliminates context size leakage
- **Pro:** Software-only solution
- **Con:** 2–8$\times$ increased latency for short prompts
- **Con:** Proportional increase in memory usage and energy

3) Decoding Normalization (Moderate Cost): Restrict all inference to a single decoding mode (e.g., always greedy or always temperature=0.7).

Trade-offs:

- **Pro:** Eliminates decoding strategy leakage
- **Pro:** Minimal performance overhead
- **Con:** Reduces model expressivity and quality for some tasks
- **Con:** May not be acceptable for general-purpose APIs

4) Noise Injection (Limited Effectiveness): Add randomized delays or dummy computation to blur timing patterns.

Trade-offs:

- **Pro:** No architectural changes required
- **Con:** Can be defeated by averaging over many observations
- **Con:** Adds latency without strong security guarantees
- **Con:** Difficult to calibrate noise level

5) Partitioning by Sensitivity: Route sensitive inference to isolated cores/machines, use shared hardware only for low-sensitivity queries.

Trade-offs:

- **Pro:** Balances security and efficiency
- **Pro:** Allows fine-grained control
- **Con:** Requires query classification infrastructure
- **Con:** May leak information through routing itself

C. Limitations

Our study has several limitations:

- **Single model:** We evaluate only TinyLLaMA-1.1B. Larger models or different architectures may exhibit different leakage characteristics.

- **Controlled experiments:** Real-world deployments involve more noise from concurrent processes, network I/O, and system events.
- **Fixed generation length:** Variable-length generation may create additional signals or reduce detection accuracy.
- **Limited prompt diversity:** Our semantic templates are deliberately controlled; realistic queries show greater variety.
- **No defenses active:** We measure baseline leakage without testing against hardened implementations.

D. Future Work

Several directions warrant further investigation:

- Evaluating attacks on larger models (LLaMA-7B, 13B) and different frameworks
- Testing robustness against noise and concurrent workloads
- Exploring cross-core and cross-VM attack scenarios
- Developing and evaluating hybrid defense mechanisms
- Investigating hardware-based mitigation (e.g., cache partitioning, predictive isolation)

VII. Related Work

A. Microarchitectural Side Channels

Cache timing attacks originated with Bernstein's AES timing analysis [9] and were formalized by Osvik et al.'s Prime+Probe technique [1]. Yarom and Falkner introduced Flush+Reload [2], which has since been applied to extract RSA keys [10], AES keys [11], and ECDSA nonces [12].

Branch predictor attacks were explored by Lee et al. [13] and Evtyushkin et al. [8], demonstrating control-flow leakage. TLB-based attacks were studied by Gras et al. [14] in the context of KASLR bypass.

Speculative execution vulnerabilities (Spectre [15], Meltdown [16]) revealed that even transient execution creates timing side channels.

B. Side Channels in Machine Learning

Neural network side-channel research has focused on several areas:

Model extraction: Oh et al. [17] used timing to steal CNN architectures. Yan et al. [5] demonstrated cache-based attacks on neural network inference.

Input inference: Chen et al. [18] showed that RNN input sequences can be partially recovered from cache timing. Duddu et al. [19] extracted training data through model inversion.

Training data membership: Shokri et al. [20] introduced membership inference attacks, later extended with confidence-based approaches [21].

Hardware accelerators: Recent work explored side channels in GPUs [22], TPUs [23], and Apple Neural Engine [24].

However, prior work has not systematically studied side-channel leakage from CPU-based LLM inference, particularly regarding high-level inference parameters like context and decoding strategy.

C. LLM Security

LLM security research has primarily focused on:

Prompt injection: Adversarial prompts that manipulate model behavior [25].

Model extraction: Stealing model weights or capabilities through black-box query access [26].

Training data extraction: Recovering training examples through carefully crafted queries [27].

Privacy attacks: Membership inference on training data [28].

Our work complements these lines by exploring microarchitectural leakage, which operates at a lower level of abstraction and cannot be mitigated by prompt filtering or output sanitization alone.

VIII. Conclusion

We presented LLMProbe, a comprehensive study of microarchitectural side-channel attacks on CPU-based LLM inference. Through systematic evaluation against TinyLLaMA-1.1B running under `llama.cpp`, we demonstrated that unprivileged co-located attackers can infer context size with 80.5% accuracy and decoding strategy with 90.0% accuracy using only SMT contention measurements. While prompt semantic classification remains challenging (33% accuracy), mathematical content shows consistently stronger leakage than other categories.

Our findings reveal that even CPU-only LLM deployments remain vulnerable to microarchitectural attacks, despite lacking the accelerator-specific optimizations that prior work has exploited. The strong leakage for context and decoding parameters creates risks for service fingerprinting, competitive intelligence, and privacy violations in multi-tenant environments.

Effective mitigation requires architectural approaches (SMT isolation, core partitioning) combined with software-level normalization (context padding, fixed decoding modes). These defenses impose non-trivial performance costs, creating a fundamental trade-off between efficiency and security for LLM inference on shared hardware.

We hope this work encourages the community to design side-channel-aware LLM inference systems and motivates hardware vendors to provide better isolation primitives for ML workloads.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-XXXXXXX and CNS-YYYYYYY.

References

[1] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[2] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium*, 2014, pp. 719–732.

[3] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in JavaScript and their implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1406–1418.

[4] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium*, 2018, pp. 991–1008.

[5] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *29th USENIX Security Symposium*, 2020, pp. 2003–2020.

[6] P. Zhang, G. Zeng, T. Wang, and W. Lu, "Tinyllama: An open-source small language model," https://github.com/jzhang38/TinyLLaMA, 2024.

[7] G. Gerganov and contributors, "llama.cpp: Port of facebook's llama model in c/c++," https://github.com/ggerganov/llama.cpp, 2023.

[8] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[9] D. J. Bernstein, "Cache-timing attacks on AES," in *Technical report*, 2005.

[10] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," in *Cryptology ePrint Archive*, 2014.

[11] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-VM attack on AES," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.

[12] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2016, pp. 242–260.

[13] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium*, 2017, pp. 557–574.

[14] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium*, 2018, pp. 955–972.

[15] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019, pp. 1–19.

[16] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium*, 2018, pp. 973–990.

[17] S. J. Oh, B. Schiele, and M. Fritz, "Towards reverse-engineering black-box neural networks," in *International Conference on Learning Representations*, 2019.

[18] Z. Chen, B. Kailkhura, and P. Varshney, "Neural network inversion in adversarial setting via background knowledge alignment," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 225–240.

[19] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, "Stealing neural networks via timing side channels," in *arXiv preprint arXiv:1812.11720*, 2018.

[20] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 3–18.

[21] A. Salem, Y. Zhang, M. Humbert, P. Berrang, M. Fritz, and M. Backes, "ML-Leaks: Model and data independent membership inference attacks and defenses on machine learning models," in *Network and Distributed System Security Symposium*, 2019.

[22] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Constructing and characterizing covert channels on GPGPUs," in *51st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2018, pp. 547–559.

[23] A. Krishna *et al.*, "Rowhammer on network-on-chip architectures," in *arXiv preprint arXiv:1911.09954*, 2019.

[24] W. Meng *et al.*, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2421–2437.

[25] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," *arXiv preprint arXiv:2211.09527*, 2022.

[26] K. Krishna, G. S. Tomar, A. P. Parikh, N. Papernot, and M. Iyyer, "Thieves on sesame street! model extraction of BERT-based APIs," in *International Conference on Learning Representations*, 2020.

[27] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models," in *30th USENIX Security Symposium*, 2021, pp. 2633–2650.

[28] Z. Song *et al.*, "Auditing privacy defenses in federated learning via generative gradient leakage," *arXiv preprint arXiv:2203.15696*, 2022.