LLM-FIN: Large Language Models Fingerprinting Attack on Edge Devices

***Abstract*—The deployment of Large Language Models (LLMs) into edge and embedded devices marks a transformative step in integrating Artificial Intelligence (AI) into real-world applications. This integration is crucial as it enables efficient, localized processing, reducing reliance on cloud computing and enhancing data privacy by keeping sensitive information on the device. In the domain of machine learning (ML) security, concealing the architecture of LLMs is imperative. Shielding the architecture protects intellectual property and thwarts malicious attempts to exploit model-specific weaknesses. Our research proposes an efficient fingerprinting method tailored to identify the architectural family of LLMs specifically within edge and embedded devices. Uniquely, our technique hinges on analyzing memory usage patterns, one of the few accessible data points in a secured edge environment. Employing a supervised machine learning classifier, our methodology demonstrates remarkable efficacy, achieving over 95% accuracy in classifying known LLMs into their architectural families. Notably, it also exhibits robust adaptability, accurately identifying previously unseen models. By focusing on memory usage patterns, our approach paves the way for a new dimension in understanding and securing AI on edge devices, balancing the need for open functionality and essential confidentiality.**

I. Introduction

The integration of Large Language Models (LLMs) into edge computing has revolutionized sectors like healthcare, autonomous vehicles, and smart homes by enabling on-device, real-time data processing, thus enhancing efficiency and data privacy [1], [2]. However, this advancement brings challenges, notably the resource limitations of edge devices and the heightened need to secure AI models against vulnerabilities and attacks [3], [4]. LLMs, critical in modern AI, face risks regarding their proprietary architecture and sensitivity in various applications. Strategies to protect these models include concealing their inner workings to prevent intellectual property theft and reduce susceptibility to attacks [5]. Fingerprinting techniques, identifying unique model characteristics, have become crucial in safeguarding AI security, especially for LLMs on edge devices [6]. These techniques detect unauthorized use and are pivotal in the evolving landscape of AI model security and integrity [7].

The landscape of fingerprinting in AI, particularly in the context of model architecture detection, has seen various innovative approaches [8]–[10]. Despite these advancements, there remain notable gaps in the existing research. Many of the current methods, such as those relying on cache-based [9] and power side-channel attacks [10], necessitate either direct physical access to the device or specific conditions like manipulation of shared resources. This requirement significantly limits the applicability of these techniques in real-world scenarios where such access is not available or feasible. Additionally, the focus on shared resources like cache or power traces often makes these methods vulnerable to simple isolation or access restriction defenses [11], [12]. These limitations highlight the need for more versatile and non-intrusive methods that can operate effectively under a broader range of conditions and against more sophisticated defense mechanisms.

In our research, we introduce a new and efficient attack method targeting prominent and cutting-edge LLM families. These models are specifically tested on NVIDIA Jetson family edge devices, known for their robust performance in edge computing. Our attack methodology is distinctive in its execution, consisting of two critical steps. Firstly, we employ the 'tegrastats' tool on an Nvidia Jetson Nano device to gather comprehensive global system traces while an LLM model is operational. This process involves meticulously selecting the most relevant features from the collected time-series data, focusing on aspects that are most indicative of the model's architecture.

The second phase of our approach involves training a machine learning classifier on this carefully labeled dataset. The classifier's objective is to accurately fingerprint the victim LLM's architecture, paying particular attention to RAM usage as a key feature. A noteworthy aspect of many model families, including those we are investigating, is the presence of variants that, while sharing a similar foundational architecture, may have adaptations such as additional layers tailored for specific tasks by developers. This diversity necessitates an examination of the transferability aspect of our attack. We delve into how effectively our attack can be applied to variants within these model families, particularly those that the attacker's classifier has not been previously trained on.

A key advantage of our method lies in its non-intrusive nature. Unlike previous methods that required manipulation of shared resources, our technique passively collects global memory-usage traces. This strategy not only enhances the feasibility of the attack in real-world scenarios but also circumvents common defensive measures such as resource isolation. Additionally, our methodology's reliance on global memory-usage data, as opposed to more sensitive or restricted data, presents a less invasive yet effective means of model architecture identification.

The efficacy of our approach is underscored by its impressive accuracy. Our results demonstrate that by analyzing a combination of GPU and CPU loads along with RAM memory usage, we can achieve an accuracy rate of 96% on trained models. This high level of precision in identifying

known models is a testament to the robustness of our method. Additionally, our experiments on transferability, which focus on the applicability of our technique to variants of model families not previously encountered by the classifier, highlight RAM usage as a pivotal factor in distinguishing between different LLM architectures. Even with unseen model variants, the attacker's classifier achieved a success rate of about 92%, further emphasizing the effectiveness of our approach.

II. Background and Related Works

Prior research has explored diverse side-channel data to achieve a range of attack goals, such as identifying model architectures [13], discerning model inputs [14], and extracting model parameters [15]. Common sources of side-channel leakage in these studies include cache [9], memory access [15], electromagnetic (EM) emissions and power consumption [16], timing information [14], and GPU statistics [17]. These sources have been foundational in advancing understanding and methodologies in the field of model security.

In a pioneering study by Duddu et al. [18], they demonstrated the feasibility of extracting a model's details by sending queries to it and monitoring the time it takes to execute these queries. This approach harnessed information about the execution duration to estimate the depth of the model. This insight substantially narrowed down the possible configurations, facilitating the accurate prediction of the targeted model's architecture.

One notable work in the GPU domain is by Wei et al. [17], who introduced an innovative attack known as $Leaky\ DNN$. This technique leverages the penalties incurred during GPU context-switching to deduce the architecture of a victim's model. A critical aspect of this approach is its reliance on access to shared GPU profiling data, which plays a pivotal role in the attack's execution. To reconstruct the layers of the Deep Neural Network (DNN), the authors employed a Denial of Service (DoS) strategy. This method deliberately slowed down the progression between the layers of the DNN model, thereby facilitating the successful extraction of the model's architectural details.

Another significant contribution is documented in the work by Patwari et al. [13]. Here, the authors put forward a method that utilizes shared memory traces from embedded GPUs on edge devices to identify a model's architectural framework. It's important to highlight, however, that this approach is specifically tailored to conventional ML models. Its applicability is somewhat limited, as it did not study the state-of-the-art models. This insight into the adaptability of the method underscores the ongoing need for security measures in the realm of LLM.

In their research, Hong et al. [8] implemented the Flush+Reload Side-Channel Attack (SCA) technique to determine the architecture of Deep Neural Networks (DNN) by monitoring specific function calls during the inference process. Similarly, Torrellas et al. [9] used both Flush+Reload and Prime+Probe methods to observe special functions, which aided in deducing the structure of the model. These approaches, however, are susceptible to cache partitioning. For successful cache-based fingerprinting attacks, precise timing mechanisms and access to a shared cache are essential. Several countermeasures against such attacks, as documented in previous studies, focus on exploiting cache behaviors to protect sensitive data from unauthorized extraction [19].

TABLE I: Comparison with state-of-the-art

Methodes	GPU profiling trace	Shared memory info	LLM aware	Access restr. resistance	Target of prediction (Accuracy)
[17]	Yes	Yes	×	×	Model (95.2%)
[8]	No	Yes	×	✓	Model (97.4%)
[9]	No	Yes	×	✓	Model (No report)
[13]	No	Yes	×	✓	Family (99%)
[20]	Yes	No	×	×	Family/Model (100%)
Our work	**No**	**Yes**	✓	✓	**Family/Model (96%)**

Power side-channel analysis attacks exploit the variable power consumption in CMOS-based digital circuits, a phenomenon closely related to the switching activity factor. Batina et al. [10], for instance, harnessed power and electromagnetic (EM) side channels to reverse engineer the architecture of models. It's important to note, though, that gathering EM or power traces for such attacks generally requires physical proximity to the targeted device. This requirement often poses a significant challenge, as physical access is not always feasible.

On the other hand, EZClone [20] offers a different approach by identifying model architectures through analyzing GPU kernel features, utilizing the PyTorch profiler. This method stands out for its resilience against isolation defense techniques. Nonetheless, it requires detailed access to shared profiling resources, making it potentially vulnerable to defenses that restrict such access.

Table I provides a comparative analysis of our research with existing studies in the field. Building upon previous studies, our research introduces an approach to classifying LLM architectures on edge devices. We focus on categorizing these applications into specific targeted model architecture families. Our method is characterized by its non-intrusive nature: it passively and remotely gathers only the global memory-usage side-channel traces from a black-box victim LLM application. This strategy eliminates the need for physical access to the device or any alterations to shared resources like the cache. We utilize these traces within a supervised learning framework, aimed at accurately classifying the network architecture. To the best of our knowledge, our research is pioneering in its attempt to utilize fingerprinting techniques for the extraction of LLM architecture information. This advancement not only contributes to a deeper understanding of model architectures but also significantly enhances the capabilities of adversarial attacks. Our approach marks a critical step forward in the realm of cybersecurity, particularly in the context of safeguarding edge-deployed deep learning applications.

III. Threat Model

Our methodology is structured within a closed-world framework, where the hypothetical attacker is assumed to have prior knowledge about the potential architectures of LLMs and the specifics of the edge device on which these models

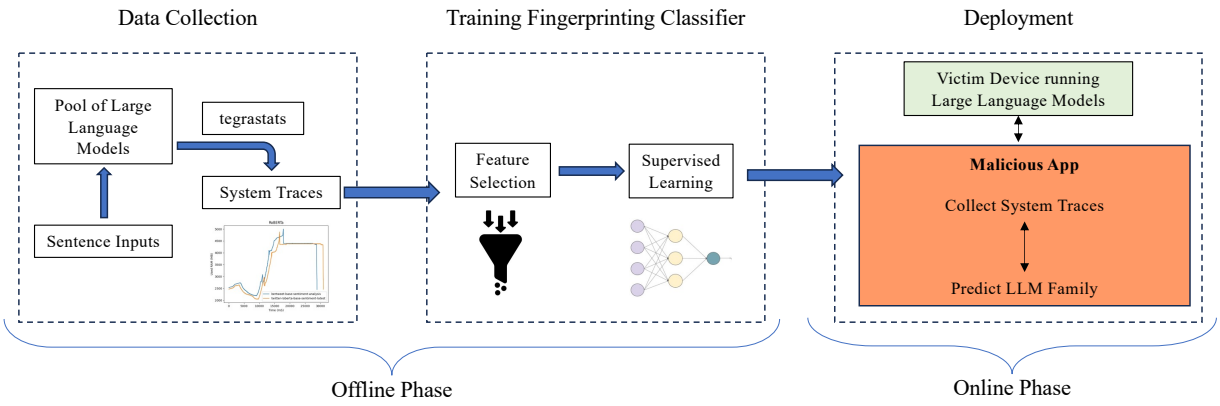

Fig. 1: Attack pipeline for the LLM architecture fingerprinting.

are operational. This advanced knowledge enables the attacker to craft a specialized, labeled dataset. This dataset is then used to train a classifier through supervised learning methods, conducted in an offline setting. The ultimate goal of the adversary in this scenario is to identify the specific LLM architecture employed by the victim ('fingerprinting'), setting the stage for more effective and targeted downstream attacks.

The foundation of our proposed fingerprinting attack rests on four key assumptions:

1) Knowledge of the Victim's Platform: The attacker is presumed to have detailed knowledge of the victim's hardware platform. This allows the attacker to replicate a similar environment, running LLMs on a comparable device to gather timing data that closely mimics the victim's device behavior.

2) Single Model Operation: It is assumed that the victim's device runs only one ML model at any given time. This assumption is crucial as it ensures that the timing data collected is solely representative of the specific model in operation, without interference from concurrent processes.

3) Familiar Model Families: The adversary's hypothesis includes the assumption that the victim's ML model belongs to one of the known model families. However, the attacker does not need prior knowledge of the model's hyper-parameters or any specific customizations. The models in question could have been fine-tuned for particular tasks, but this does not deter the fingerprinting process.

4) Adversary's Privilege Level: The attacker operates with user-space privileges, commonly referred to as Ring 3 privileges. This level of access implies that the adversary does not have deep system-level control but can execute standard user-level operations, which are sufficient for conducting the fingerprinting attack.

These assumptions create a structured environment in which our attack methodology can be effectively applied, allowing for the precise identification of LLM architectures and paving the way for more sophisticated adversarial strategies.

IV. Fingerprinting Methodology

A. Attack Overview

In our study, we focus on the use of resource utilization information as a tool to fingerprint prominent LLM architectures, particularly those sourced from *Hugging Face* [21]. A standard Machine Learning model inference pipeline typically encompasses several key steps:

Loading LLM Hyperparameters: This step involves initializing the model with its predefined settings and configurations, which dictate its behavior and performance. Loading Inputs for the LLM: Here, the specific data or inputs that the LLM will process are loaded into the system. Inference Task: This critical phase involves conducting a forward pass of the input through the model to generate outputs. It's where the actual 'thinking' or processing of the model occurs. Our research is visually summarized in Figure 1, which outlines the structure of our proposed LLM fingerprinting attack. This approach is divided into two distinct phases:

1) Offline Phase: In this preliminary phase, the adversary observes the victim's LLM operations, capturing relevant information. The data gathered during monitoring is then processed (through steps like normalization and smoothing) to create a refined, labeled dataset. This dataset serves as the foundation for training a classifier specifically designed for fingerprinting purposes.

2) Online Phase: This is the phase where the actual attack is carried out. The attacker utilizes the classifier trained during the Offline phase. This classifier is now adept at real-time fingerprinting of the LLM architecture. By implementing this classifier, the attacker can accurately identify the architecture and family of the ML model in use during runtime.

By adopting this two-phase strategy, an attacker is positioned to effectively conduct a runtime fingerprinting attack. This attack is aimed at pinpointing the specific LLM architecture and family, providing crucial insights for more targeted and potent downstream adversarial actions.

B. LLM Selection

To construct our comprehensive offline fingerprinting database, we employed the Hugging Face library along with its inference pipeline. This approach enabled us to gather precise timing data for a selection of widely-used Hugging Face pretrained LLMs. The specific models we chose for this study are detailed in Table II. These models are primarily designed for tasks like sentence similarity assessment or text

classification. Each model we selected was categorized into distinct model families. This categorization was based on their respective architectures as stated in the Hugging Face model cards or as found on the models' official websites. For instance, some of the model families we identified include BERT and DialogRPT. This classification of LLMs into model families is a strategic step in our research. It allows us to develop and train a Machine Learning (ML) classifier with a specific focus. The purpose of this ML classifier is to be a critical tool in the subsequent online attacking phase.

In the online attacking phase, the trained ML classifier's role is to efficiently extract timing information from a target device. This device is presumed to be running an LLM program that the classifier has not previously encountered. The classifier's objective is to accurately determine the model family of the unseen LLM program based on the timing information. This identification is a pivotal element in our research, as it provides essential information that can significantly enhance the effectiveness of subsequent adversarial attacks. By understanding the model family of the LLM running on the victim's device, we can tailor our adversarial strategies more precisely, increasing the likelihood of a successful attack.

C. Inference Pipeline Design

Our inference pipeline is meticulously structured, incorporating fundamental components that are consistent across various LLM inference applications. These essential components include the loading of data, the importation of model weights, and the execution of inference tasks on data streams. Specifically, the pipeline processes the hate speech dataset, comprising 10,945 entries sourced from Stormfront forum posts.

The design of the inference pipeline, however, exhibits slight variations when applied to text classification models compared to sentence similarity models. In the case of text classification models, our approach leverages the pipeline abstraction layer offered by the $transformers$ API. This layer facilitates the loading of pretrained models from the Hugging Face library. Subsequently, these models undertake sequential inference on a subset of 2,000 sentences extracted from the dataset.

Conversely, the inference pipeline for sentence similarity models integrates the $SentenceTransformers$ API for model loading. This pipeline computes tensor encodings for 100 distinct batches, each containing 100 sentences. Following this, it calculates the $cosine$ similarity for sentences within each batch, ultimately deriving a sentence similarity score for every pair within the same batch.

The rationale behind using different sentence quantities for the two model types stems from the observation that text classification models generally exhibit slower inference speeds compared to sentence similarity models. Our objective is to standardize the duration of both the model loading and inference phases across varied models as much as possible. This standardization ensures a more uniform comparison and analysis.

TABLE II: Pool of LLM models used in Model set 1

Model Set 1	**Family**	**Task**
all_datasets_v4_MiniLM-L12	MiniLM	Sentence similarity
all-MiniLM-L12-v2		
all-mpnet-base-v2	MPNet	
multi-qa-mpnet-base-cos-v1		
gtr-t5-base	T5	
sentence-t5-base		
finbert	BERT	Text classification
bart-base-mnli		
twitter-emotion-deberta-v3-base	DeBERTa	
DialoRPT-depth	DialogRPT	
distilbert-base-uncased-finetuned-emotion	DistillBERT	
bertweet-base-sentiment-analysis	RoBERTa	
roberta-hate-speech-dynabench-r4-target		

Furthermore, we opted for a batch size of 100 sentences for sentence similarity models. This decision is important, as it facilitates the calculation of similarity scores among a larger array of sentence pairs. Such a setup more accurately reflects the real-world application scenarios of sentence similarity models, where they frequently process extensive sets of sentence pairs to discern their comparative similarities.

D. Data Collection

In our research, we employ the 'tegrastats' utility as a key tool for data collection during the operation of LLM inference pipelines. The process for each pretrained LLM we investigate is methodically structured as follows:

1) Initiating Tegrastats: Before running any LLM inference code, we initiate the tegrastats program as a background process. This setup ensures that tegrastats is actively monitoring and recording the system's performance metrics from the very start of the LLM's operation.

2) Running LLM Inference Code: Once tegrastats is up and running, we proceed to execute the LLM inference code. During this phase, tegrastats continues to collect detailed system data, capturing the dynamics of the device as it processes the LLM tasks.

3) Stopping Tegrastats and Data Storage: Immediately after the completion of the LLM inference code execution, we halt the tegrastats program. The collected data, which effectively represents a time series log of the system's performance during the LLM operation, is then saved as a text file. This file serves as a detailed record of that specific execution's system metrics.

4) Repetitive Execution and Data Collection: Each LLM's inference pipeline is executed a total of 45 times to gather comprehensive data. However, to ensure quality and consistency, we only retain the tegrastats logs from the last 40 executions. These logs collectively form our time series dataset.

5) High-Resolution Data Sampling: To achieve the finest granularity in our data, the tegrastats program is configured to sample hardware information at its maximum capability, which is a rapid 1 millisecond (ms) sampling rate. This high-resolution sampling allows us to capture the most detailed and accurate representation of the system's performance during LLM operations.

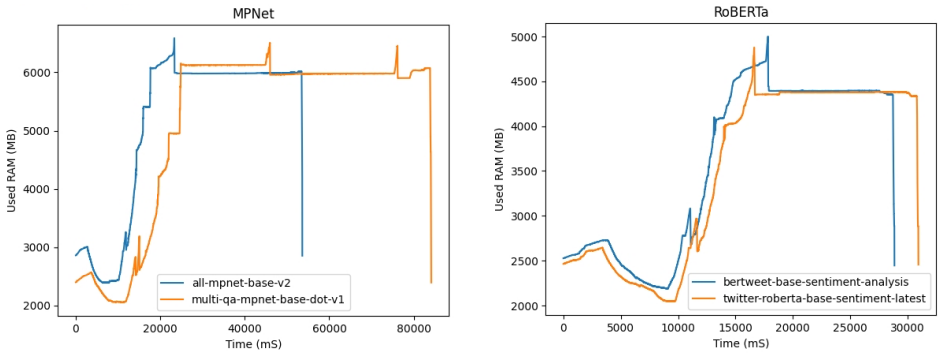

Fig. 2: Comparative overview of total RAM utilization for two LLM families, monitored using Tegrastats on Jetson Nano. Memory consumption trends show distinctions among different Families. For illustrative purposes, the time axis is not scaled.".

E. Training the Fingerprinting Classifier

The primary goal of our fingerprinting classifier is to utilize resource utilization data to accurately predict the architecture and family of Large Language Models (LLMs) when they are operational on edge devices. To accomplish this classification task, we have employed the $Sktime$ Python library [22], a recent and valuable addition to the open-source ecosystem. $Sktime$ is notable for its compatibility with scikit-learn and its specialized capabilities for handling time series data.

Within the $Sktime$ framework, we opted to use the $ROCKET$ model (Random Convolutional Kernel Transform), as referenced in Dempster et al. [23]. The $ROCKET$ model is renowned for its high accuracy and efficiency in processing time series data.

The dataset required for the $ROCKET$ model is structured in a specific format:

1) Input Samples: The data should be arranged in a three-dimensional numpy array, with the dimensions representing (instance, feature, time point). Each $instance$ in this array corresponds to an individual sample, and each sample comprises multiple features captured over various time points. In our context, each resource usage trace (like CPU and Memory utilization) is treated as a separate feature. 2) Labels: The labels for these input samples are provided in a one-dimensional numpy array. Each element in this label array corresponds to the class or category of the LLM architecture for the respective sample in the input data.

This data format is particularly suited for processing by the $Sktime$ library. For the $ROCKET$ model, we configure a hyperparameter: the number of random convolutional kernels, which we set to 3,000. This setting is pivotal for the model's performance.

To ensure the classifier is robust and can generalize well, especially considering it will be used on the same device as the victim's but with potential minor variations, we implement a normalization step. Resource usage values are normalized to a range between 0 and 1. This normalization process is crucial as it allows our classifier to focus on identifying patterns or shapes in the data, rather than being influenced by raw numerical values.

V. Evaluation

A. Experimental setup

1) Device Specification: For our experiment, we utilized the Jetson Xavier NX device. This device is built with a 6-core NVIDIA Carmel processor based on ARM architecture, features a Volta GPU with 384 CUDA cores and 48 Tensor cores, and is equipped with 8GB of 128-bit LPDDR4x memory. The Jetson Xavier NX boasts an AI task capability of up to 21 TOPS, enhanced by its robust components. We set up the environment of the Jetson Xavier NX using Jetpack SDK version 5.1. This SDK includes Jetson Linux 35.2.1, based on Ubuntu 20.04, and integrates CUDA 11.4. The Jetson Xavier NX offers flexible power mode settings, including 10W, 15W, and 20W, which correspondingly affect the performance of its CPU and GPU. In the 10W mode, the GPU operates at 800MHz, while in the 15W and 20W modes, it reaches up to 1100MHz. The CPU in the Jetson Xavier NX allows for adjustable core activation: in 10W mode, it can be set to 2 cores at 1.5GHz or 4 cores at 1.2GHz; in 15W and 20W modes, it can operate with 2 cores at 1.9GHz, or 4 to 6 cores at 1.4GHz. Our fingerprinting experiments were conducted using the default 15W power mode with 2 active CPU cores.

B. Classifier Accuracy

To evaluate the adaptability of our attack and its proficiency in classifying models not included in the training set, we introduced an additional collection of models from the initially chosen families (termed Model Set 2, as shown in Table III). The efficacy of our classifier on both the initial group (Model Set 1) and this new set (Model Set 2) is outlined in Table IV, where we present the accuracy metrics for model and family classification.

Figure 2 provides a visual representation of four distinct data examples, fed into our classifier. Each example illustrates RAM utilization (depicted on the Y-axis) of the edge device, corresponding to different LLM models over consecutive time points (shown on the X-axis). The accuracy of model classification is relevant exclusively to Model Set 1, which includes models used in the training phase. Our results show that the classifier achieved an accuracy exceeding 96% in

TABLE III: Pool of unseen LLM models used in Model set 2

Model Set 2	Family	Task
multi-qa-MiniLM-L6-cos-v1	MiniLM	Sentence similarity
multi-qa-mpnet-base-dot-v1	MPNet	
sentence-t5-large	T5	
bart-fined-tuned-on-entailment-classification	BART	Text classification
deberta-base-mnli	DeBERTa	
DialoRPT-updown	DialogRPT	
Text_classification_model_1_pytorch	DistillBERT	
twitter-roberta-base-sentiment-latest	RoBERTa	

TABLE IV: Pool of unseen LLM models used in Model set 2

	Family classification			Model Classification		
Feature	*CPU*	*Memory*	*CPU+Mem*	*CPU*	*Memory*	*CPU+Mem*
Set 1	75.6	91.2	96.7	68.7	89.4	92.0
Set 2	66.5	88.1	92.3	—	—	—

identifying the family of the architectures for the models in Model Set 1. Impressively, it also accurately classified the family of the previously unseen models in Model Set 2 with an average accuracy of 92%. These results validate our classifier's capability in accurately categorizing both familiar and novel model families, highlighting its transferability and practicality.

Our analysis indicates that the MiniLM family models exhibit the highest classification accuracy, while the DialogRPT family models are less accurately identified. Remarkably, the MPNet architecture stands out for its distinctiveness, ranking as the second most accurately classified model. We also observed a consistent pattern of misclassification between models RoBERTa and DistillBERT, a finding that aligns with expectations considering their similar model characteristics.

It's worth noting that ongoing refinement and optimization of the classifier could address some of the observed misclassifications, thereby improving its accuracy and robustness in identifying ML model architectures. The high classification precision for Model Set 2 emphasizes the attack's generalizability, ensuring effectiveness even against models not present in the training set. Such a feature enhances the real-world applicability of our approach, particularly in scenarios where adversaries might encounter unfamiliar ML models.

VI. Conclusion

This paper has presented an approach to fingerprinting LLM architectures deployed on edge devices, addressing a critical gap in the field of edge AI security. Our method, characterized by its non-intrusive nature, successfully leverages global memory-usage side-channel traces to identify model architectures, thereby circumventing the limitations of previous techniques of manipulation of shared resources. Our research demonstrates that the combination of GPU memory and CPU loads can be effectively used to achieve a remarkable accuracy rate of 96.7% in identifying trained model architectures. Such findings not only validate the efficacy of our approach but also highlight the significance of memory usage patterns as a reliable indicator in LLM architecture identification. We believe this is a valuable insight for safeguarding intellectual property and enhancing the security of AI models against adversarial attacks. As the deployment of AI on edge devices continues to grow, the importance of robust security measures such as the one proposed in this paper becomes increasingly paramount.

References

[1] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, A. Asmita, R. Tsang, N. Nazari, H. Wang *et al.*, "Large language models for code analysis: Do llms really do their job?" *arXiv preprint arXiv:2310.12357*, 2023.

[2] Y.-Z. Lin, M. Mamun, M. A. Chowdhury, S. Cai, M. Zhu, B. S. Latibari, K. I. Gubbi, N. N. Bavarsad, A. Caputo, A. Sasan *et al.*, "Hw-v2w-map: Hardware vulnerability to weakness mapping framework for root cause analysis with gpt-assisted mitigation suggestion," *arXiv preprint arXiv:2312.13530*, 2023.

[3] N. Nazari, C. Fang, S. M. PD, and H. Homayoun, "Don't cross me! cross-layer system security," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–2.

[4] H. M. Makrani, Z. He, S. Rafatirad, and H. Sayadi, "Accelerated machine learning for on-device hardware-assisted cybersecurity in edge platforms," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2022, pp. 77–83.

[5] N. Nazari, H. M. Makrani, C. Fang, B. Omidi, S. Rafatirad, H. Sayadi, K. N. Khasawneh, and H. Homayoun, "Adversarial attacks against machine learning-based resource provisioning systems," *IEEE Micro*, 2023.

[6] H. Sayadi, Y. Gao, H. Mohammadi Makrani, J. Lin, P. C. Costa, S. Rafatirad, and H. Homayoun, "Towards accurate run-time hardware-assisted stealthy malware detection: a lightweight, yet effective time series cnn-based approach," *Cryptography*, vol. 5, no. 4, p. 28, 2021.

[7] H. M. Makrani, H. Sayadi, N. Nazari, K. N. Khasawneh, A. Sasan, S. Rafatirad, and H. Homayoun, "Cloak & co-locate: Adversarial railroading of resource sharing-based attacks on the cloud," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2021, pp. 1–13.

[8] S. Hong *et al.*, "Security analysis of deep neural networks operating in the presence of cache side-channel attacks," *arXiv*, 2018.

[9] M. Yan *et al.*, "Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures," in *29th USENIX Security Symposium*, 2020.

[10] L. Batina *et al.*, "Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel," in *28th USENIX Security Symposium*, 2019.

[11] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "{PKU} pitfalls: Attacks on {PKU-based} memory isolation systems," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1409–1426.

[12] G. S. Mahmood, D. J. Huang, and B. A. Jaleel, "A secure cloud computing system by using encryption and access control model," *Journal of Information Processing Systems*, vol. 15, no. 3, pp. 538–549, 2019.

[13] K. Patwari *et al.*, "Dnn model architecture fingerprinting attack on cpu-gpu edge devices," in *7th EuroS&P*. IEEE, 2022.

[14] G. Dong *et al.*, "Floating-point multiplication timing attack on deep neural network," in *SmartIoT*. IEEE, 2019.

[15] W. Hua *et al.*, "Reverse engineering convolutional neural networks through side-channel information leaks," in *55th DAC*, 2018.

[16] Y. Xiang *et al.*, "Open dnn box by power side-channel attack," *IEEE Transactions on Circuits and Systems II*, vol. 67, no. 11, 2020.

[17] J. Wei *et al.*, "Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel," in *50th DSN*. IEEE, 2020.

[18] V. Duddu *et al.*, "Stealing neural networks via timing side channels," *arXiv preprint arXiv:1812.11720*, 2018.

[19] D. Page, "Defending against cache-based side-channel attacks," *Information Security Technical Report*, vol. 8, no. 1, pp. 30–44, 2003.

[20] J. Weiss *et al.*, "Ezclone: Improving dnn model extraction attack via shape distillation from gpu execution profiles," *arXiv*, 2023.

[21] *https://huggingface.com*.

[22] *https://github.com/sktime/sktime*.

[23] A. Dempster *et al.*, "Rocket: exceptionally fast and accurate time series classification using random convolutional kernels," *Data Mining and Knowledge Discovery*, vol. 34, no. 5, 2020.