Enabling DVFS Side-Channel Attacks for Neural Network Fingerprinting in Edge Inference Services

Enabling DVFS Side-Channel Attacks for Neural Network Fingerprinting in Edge Inference Services

Erich Malan, Valentino Peluso, Andrea Calimera, Enrico Macii*
Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy
*Interuniversity Department of Regional and Urban Studies and Planning, Politecnico di Torino, Turin, Italy
{erich.malan, valentino.peluso, andrea.calimera, enrico.macii}@polito.it

***Abstract*—The Inference-as-a-Service (IaaS) delivery model provides users access to pre-trained deep neural networks while safeguarding network code and weights. However, IaaS is not immune to security threats, like side-channel attacks (SCAs), that exploit unintended information leakage from the physical characteristics of the target device. Exposure to such threats grows when IaaS is deployed on distributed computing nodes at the edge. This work identifies a potential vulnerability of low-power CPUs that facilitates stealing the deep neural network architecture without physical access to the hardware or interference with the execution flow. Our approach relies on a Dynamic Voltage and Frequency Scaling (DVFS) side-channel attack, which monitors the CPU frequency state during the inference stages. Specifically, we introduce a dedicated load-testing methodology that imprints distinguishable signatures of the network on the frequency traces. A machine learning classifier is then used to infer the victim architecture. Experimental results on two commercial ARM Cortex-A CPUs, the A72 and A57, demonstrate the attack can identify the target architecture from a pool of 12 convolutional neural networks with an average accuracy of 98.7% and 92.4%**

***Index Terms*—Inference-as-a-Service, Convolutional Neural Networks, Side-Channel Attacks, DVFS.**

I. Introduction

Inference-as-a-Service (IaaS) is a licensing and delivery model that lets connected clients get access to pre-trained deep neural networks while preserving the intellectual property rights of the service provider. Originally conceived for the cloud, the IaaS paradigm can be implemented on edge nodes, like IoT gateways, to improve response times, scalability, and geographical coverage [1]. The edge-IaaS infrastructure relies on the microservice architectural approach. Microservices leverage virtualization technologies to build hardware-agnostic inference engines deployable on a multitude of computing platforms, including embedded systems commonly equipped with low-power CPUs, such as the ARM Cortex-A cores. The clients query the inference service through an Application Programming Interface (API) that prevents access to the neural network, protecting code and weights. Unfortunately, microservices deployed on distributed edge nodes are more vulnerable to malicious acts [2], like Side-Channel Attacks (SCAs), which can extract sensitive information about the neural network from the physical characteristics of the hosting device.

979-8-3503-1175-4/23/$31.00 ©2023 IEEE

TABLE I
Overview of black-box SCAs against IaaS.

Reference	Access	Target Device	Side-Channel Signal	Information
[3]	Physical	ARM Cortex-A	Power consumption	Architecture
[4]	Physical	NVIDIA GPU	Memory and PCIe bus	Architecture
[5]	Physical	NVIDIA GPU	PCIe bus	Weights
[6]	Physical	ARM Cortex-M	EM emission	Weights
[7]*	Remote	Intel x86	Cache timing	Hyperparams
[8]	Remote	Intel x86	Latency	# of Layers
[9]	Remote	Intel x86	Performance counters	Layers Type
[10]‡	Remote	NVIDIA Tegra	Resource utilization	Family
[11]*	Remote	Intel x86	Cache timing	Architecture
[12]†	Remote	NVIDIA GPU	Power consumption	Architecture
This work	Remote	ARM Cortex-A	DVFS state	Architecture

* Manipulate the execution flow. †Require integrated power sensors.
‡ Neural networks belonging to the same family show the same topology and core organization but differ in the number of layers and/or filters.

Previous studies demonstrated various types of SCAs that target different hardware platforms and can extract different details. Table I summarizes state-of-the-art SCAs, and in particular, black-box SCAs, namely, methods that assume no prior knowledge of the deep neural network. The table offers a taxonomy based on four main features: the required access, the target device, the source of side-channel leakage, and the recovered information. Physical attacks require direct access to the target device for measuring physical signals, like power consumption [3], memory and PCIe bus traces [4], [5], or electromagnetic emissions (EM) [6]. Those signals contain fine-grained information from which the attacker can retrieve the neural network architecture and even reconstruct the network weights, enabling the replica of the neural network. However, physical probing and costly laboratory equipment limit applicability in real-life scenarios. On the other hand, remote attacks are software methods based on monitoring system metrics and performance counters accessible from operating system logs without special privileges. However, they can simply retrieve partial information, like a subset of architectural hyperparameters (e.g., stride, pooling, padding in case of convolutional neural networks) [7], the number of layers [8], the type of layers (without details about the number of filters) [9], the family to which the neural architecture belongs [10]. Very few remote attacks allow for the identification of the neural architecture [11], [12]. Moreover, remote attacks show several weaknesses that limit their detection capability (64.17% Top-1 accuracy, as reported in [12]). For example, cache attacks [11] require the ability to modify the execution flow through additional cache instructions, which

may interfere with the network execution. Alternatively, attacks that probe power consumption using software-readable on-chip current sensors are affected by low sampling rates and low resolution of the sensors [13]. To notice that those current sensors are mounted on development kits but are often unavailable in production-grade boards.

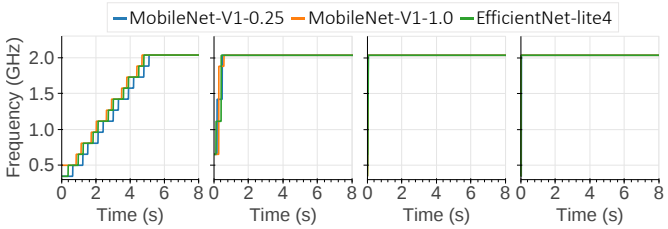
This work uncovers a vulnerability in ARM Cortex-A CPUs that can be exploited to bypass IaaS protection via remote SCA. Specifically, our study investigates the use of Dynamic Voltage and Frequency Scaling (DVFS) state as a signal for neural network fingerprinting. We demonstrate that existing DVFS SCAs are ineffective at extracting signatures that could lead to the reconstruction of the neural architecture. Hence, we introduce a novel method based on a load-testing procedure that sends a sequence of queries to the prediction API according to pre-defined schemes. The procedure regulates the service load inducing variations in the voltage-frequency state of the device under attack. The generated frequency traces are processed through a machine learning (ML) classifier trained offline to infer the neural architecture. The proposed methodology is validated on two embedded CPUs from the ARM Cortex-A family: the A72 and A57. The collected results show that our approach achieves an average accuracy of 98.7% (A72) and 92.4% (A57) in detecting the neural architecture from a pool of 12 state-of-the-art convolutional neural networks (CNNs) suited for embedded applications. These findings suggest that DVFS can be exploited as a source of leakage for neural architecture theft, highlighting the need for security measures against these types of attacks.

II. Background & Related Works

A. DVFS & Governors

DVFS is a runtime power and thermal management technique for digital cores. Low-power ARM CPUs based on the Cortex-A architecture have a predefined set of voltage-frequency pairs available, and each voltage level is associated with one specific operating frequency. The *CPUFreq* is the software interface integrated into the Linux kernel in charge of controlling DVFS. It consists of two main components: the scaling governors and the scaling driver. Scaling governors implement algorithms to set up the right CPU frequency based on the system's workload. The scaling driver is responsible for accessing the hardware interfaces to change the CPU frequency as requested by the scaling governors. The standard workflow of a governor involves continuous monitoring of the CPU utilization, computed as the ratio between the number of busy (non-idle) CPU time and the total time elapsed since the last evaluation. The time interleaving two consecutive workload estimations defines the sampling period, which can range from microseconds to milliseconds.

The Linux kernel embeds four standard scaling governors, i.e., *conservative*, *ondemand*, *schedutil*, *interactive* [14], differently available depending on the hosting platform. The *conservative* policy gradually adjusts the frequency level following a hysteresis scheme. When the CPU utilization exceeds a predefined upper threshold (e.g., 80%), the operating frequency



(a) conservative (b) ondemand (c) schedutil (d) interactive

Fig. 1. Examples of frequency traces collected on A57 with the standard procedure for DVFS SCAs.

is increased to the next available level; when it falls below the lower threshold (e.g., 20%), the frequency is decreased to the previous level. The *ondemand*, *schedutil*, and *interactive* governors implement a workload-proportional scaling approach, where the frequency is set to values linearly proportional to the CPU utilization. The *ondemand* policy maps 0%/100% CPU utilization to the lowest/highest frequency available, whereas the *schedutil* policy applies an over-scaling factor of 1.25× to prevent the frequency from being too low, which could lead to poor performance. Furthermore, *schedutil* boosts the frequency to the maximum level in case of a deadline event. The *interactive* governor favours frequency up-scaling and penalizes down-scaling. It reacts to sudden increases in CPU utilization by instantly forcing the maximum frequency level, and it waits a short period before decreasing the frequency to prevent costly performance fluctuations. That makes the *interactive* governor more responsive to intensive workloads but also more conservative in saving power.

Users with root privileges can alter the governor settings and parameters, e.g. the thresholds. Generic users without special permissions are limited to reading the governor policy and monitoring the operating frequency.

B. DVFS Side-Channel Attacks

DVFS SCAs collect frequency traces from the victim device and use ML models trained offline to steal sensitive data or to detect the applications in use. Extensive investigations on the efficacy of DVFS SCAs for data theft have been conducted in [15], showing that it is possible to identify the websites visited by users and even retrieve passwords entered on their smartphones. In [16] and [17], the authors showed the use of DVFS SCAs in the context of application detection. The general idea relies on profiling DVFS states during continuous loops of the same application. The approach was validated in [16] for a collection of 22 Android benchmarks running on an ARM Cortex-A CPU. A more exhaustive assessment conducted in [17] presented a benchmarking over two representative application suites: *PARSEC 3.0* and *SGX-bench*. The report proved that DVFS SCAs are effective for applications with long execution times and varying CPU usage (PARSEC 3.0) but are prone to fail for short and intensive workloads (SGX-bench), where the F1 score reaches a mere 10% over ten applications. As discussed in the next section, neural networks fall into the latter category, which motivates this work.

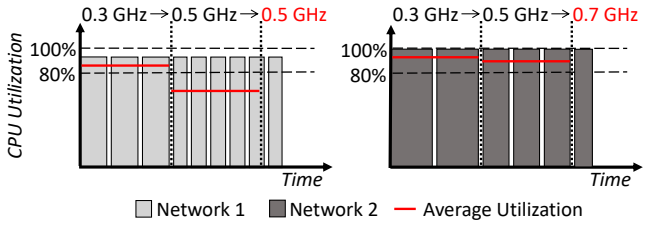


Fig. 2. CPU Frequency evolution during the execution of two different networks with the proposed approach.

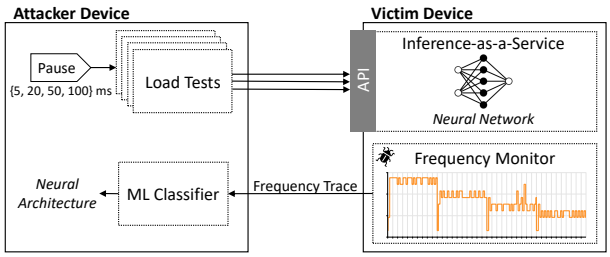


Fig. 3. Extraction phase overview.

III. Methodology

A. Understanding the Limitations of DVFS SCAs

In standard DVFS SCAs, the target application is run repeatedly while monitoring how the CPU frequency evolves. The same approach can be applied to IaaS by making continuous requests to the prediction API. Unfortunately, the deep neural networks backed by embedded applications are optimized through extensive algorithmic and code optimizations [18], resulting in dense workloads that are short in time and highly resource-demanding. This forces DVFS governors to set the highest frequency available and keep it constant until the end of the inference, no matter the network architecture. We deployed three networks of different sizes and algorithmic complexity (i.e., MobileNet-V1-0.25, MobileNet-V1-1.0, EfficientNet-lite4) on an A57 CPU (more technical details in Sec. IV-A) and we profiled the frequency traces during 8 seconds of continuous queries using the four governors introduced in Sec. II-A. The results reported in the line plots of Fig. 1 show overlaps that prevent discriminating the neural architecture under attack. The high resource demand quickly drives and keeps the frequency to the maximum level (2.0 GHz) even for the *conservative* governor.

B. Altering the Frequency Profiles

We propose a simple yet effective strategy to alter the CPU frequency updates driven by DVFS governors. The idea is to introduce pause intervals between consecutive API requests. Longer (shorter) idle periods result in lower (higher) CPU utilization and eventually lower (higher) frequency levels. The number of such pauses will be inversely proportional to the response time, a metric tightly coupled with (i) the neural architecture deployed for serving the request and (ii) the CPU frequency. Different frequency profiles over long observation periods serve as a signature to discriminate the neural architecture.

Fig. 2 graphically depicts how different frequency states could originate from two different neural architectures (Network 1 and Network 2, with Network 1 less resource-demanding than Network 2). For the sake of simplicity, the example assumes a *conservative* governor: the frequency is increased when the average CPU utilization during the observation period exceeds 80%. The coloured bars (light grey for Network 1, dark grey for Network 2) refer to the processing stage of an inference request. The bars' width reflects the CPU time to process the request, which is a function of the neural architecture and the CPU frequency. The bars' height reflects the CPU utilization during the request processing, which is another metric dependent on the neural architecture. Both response time and CPU utilization of each inference request affect the workload estimation during an observation window. At the end of each window, indicated by the vertical dotted lines, the governor evaluates the average CPU utilization (red lines in the plots) and decides on the frequency update. In the first window, the CPU operates at the lowest frequency of 0.3 GHz for both networks. Since the average CPU utilization exceeds the 80% threshold, the governor increases the frequency to 0.5 GHz for both networks. In the second window, the pause intervals (white spaces in the plots) inserted between the requests alter the workload estimation. With the increased frequency, the requests are processed faster (thinner bars for both networks). The number of pauses also increases (more white spaces), leading to a decrease in average CPU utilization, potentially falling below the threshold. This is the case for the lighter network (Network 1), for which the CPU utilization falls below the threshold ($<$80%) and the governor does not change the frequency. The same is not true for the more complex network (Network 2), where the average utilization is still over-threshold ($>$80%) and the governor drives a frequency increase to 0.7 GHz. Therefore, the frequency traces for the two networks exhibit distinct patterns. Without the pause intervals, the governor would have made the same frequency update for both networks in the third window, resulting in no discernible difference between their frequency traces. It is worth noticing that the length of the pause intervals is a hyper-parameter that can be leveraged to enhance the differentiation capability among different neural networks and CPUs. Further implementation details are provided in the following subsections.

C. Load testing

Load testing is a type of performance testing that aims at assessing how a system behaves under (i) specific workloads or (ii) requests volume. A standard approach is to generate a large amount of traffic by creating a pool of concurrent virtual users that query the service according to a predefined request rate. The number of virtual users and the request rate are parameters that can be fine-tuned to emulate different scenarios and test the stability and scalability of the service.

We borrowed a typical load testing infrastructure to implement the pausing mechanism introduced in the previous

TABLE II
HARDWARE PLATFORMS, OPERATING SYSTEM (OS), MAXIMUM (MAX. FREQ.) AND MINIMUM (MIN. FREQ.) FREQUENCY, AND NUMBER OF FREQUENCY LEVELS OF THE TARGET CPUS.

CPU	Platform	OS	Min. Freq.	Max. Freq.	Levels
A72	Raspberry Pi 4	Ubuntu 22.04	600 MHz	1.500 GHz	10
A57	NVIDIA TX2	Ubuntu 18.04	345 MHz	2.035 GHz	12

TABLE III
GOVERNORS AND THEIR SAMPLING PERIOD (ms) ON THE TARGET CPUS.

CPU	conservative	ondemand	schedutil	interactive
A72	10.0	10.0	10.0	N/A
A57	300.0	300.0	0.5	20.0

TABLE IV
POOL OF NEURAL NETWORK ARCHITECTURES.

Architecture	Layers	Size (MB)	Latency (ms)		Util. (%)	
			A72	A57	A72	A57
MobileNet-V1-0.25	28	0.49	4.71	3.57	85.6	92.0
MobileNet-V1-0.50	28	1.31	9.68	7.44	91.9	95.1
MobileNet-V1-0.75	28	2.51	17.80	13.53	95.0	97.0
MobileNet-V1-1.0	28	4.09	27.86	21.03	96.8	98.0
MobileNet-V2	53	3.42	23.39	17.85	96.2	97.7
MobileNet-V3-small	54	2.52	10.20	8.06	92.5	95.6
MobileNet-V3-large	64	5.35	25.80	20.26	96.4	97.9
EfficientNet-lite0	50	5.18	21.62	16.48	90.7	90.3
EfficientNet-lite1	65	6.12	32.32	23.87	92.4	92.6
EfficientNet-lite2	65	6.83	44.96	33.37	93.6	93.6
EfficientNet-lite3	74	9.15	68.97	51.46	95.2	95.2
EfficientNet-lite4	92	14.34	119.60	89.04	96.8	96.9

subsection, specifically a single virtual-user setup with the pause interval as the only testing parameter. The pause interval is defined as the time distance between the service response and the next client request and can be used as a knob to improve the SCA efficiency. More in detail, it is a knob to increase and reduce CPU utilization. The optimal values, i.e., those which produce enough frequency variations, may depend on non-controllable factors, such as the governor in use at the endpoint hosting the inference service and the neural network characteristics (latency and utilization). We thereby resorted to multiple load tests where the pause length is swept over a predefined set of values, as detailed in the next subsection.

D. Attack Overview

The SCA plan involves performing four load tests in sequence, each with a duration of 10 s and a growing pause length of 5, 20, 50, and 100 ms.

We consider a threat model based on the following assumptions. First, the attacker owns a device, referred to as the template device, with characteristics and specifications identical to those of the victim device; this falls into the broad category of *profiled attacks* [19]. Second, the attacker has access to the victim's device to install malware for monitoring and transmitting the CPU frequency traces. This kind of malware can gather raw data without requiring root permissions. Third, the target neural network belongs to a pool of well-known neural network architectures (although network weights may differ). This assumption is quite realistic in most use cases. It is standard practice today to leverage transfer learning [20], a deep learning strategy where a neural architecture is taken from a collection of publicly available networks and fine-tuned on a proprietary dataset.

Following the standard organization of a profiled attack, our method consists of a *profiling* phase and an *extraction* phase.

In the profiling phase, we collect frequency traces of the target neural architectures to train an ML classifier capable of inferring the target architecture from traces. To collect the dataset, we execute the load tests described in Sec. III-C on each target neural architecture deployed on the template device. During the execution of the load tests, a background process records the frequency profiles by reading the frequency level every 10 ms using the *CPUFreq* interface. The same load test is processed 50 times to account for small fluctuations in the frequency traces caused by operating system routines running in the background. This procedure is repeated four times, one for each pause length. The resulting dataset thus includes multiple frequency traces annotated with a label corresponding to one neural architecture. The classifier consists of a MiniRocket [21] feature extractor, commonly used for time-series data, followed by a multinomial logistic regressor; it is trained in a supervised manner using the *lbfgs* solver and 5-fold stratified cross-validation for hyper-parameters optimization. The training loop consists of 200 iterations. It is worth emphasizing that the data collection campaign and the training pipeline are repeated for all governors and template devices considered in our experiments, resulting in a dedicated ML classifier for each governor and hardware configuration.

In the extraction phase, the attack is deployed to steal the target neural architecture, as illustrated in Fig. 3. The victim device is one of the template devices analyzed in the profiling phase, but the target neural architecture is unknown. The attacker runs the four load tests on the prediction endpoint, recording the victim device's CPU frequency through the pre-installed malware. After the load tests, the collected frequency trace and the active governor are sent to the attacker and fed into the ML classifier trained for that specific governor. The ML classifier returns the neural architecture used by the victim inference service.

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

We operated two commercial platforms as test benches: the Raspberry Pi 4 and the NVIDIA Jetson TX2. These platforms are powered by different versions of the ARM Cortex-A CPU, i.e., A72 and A57 respectively, whose technical specifications are reported in Table II. Both CPUs are quad-core but differ in terms of frequency range, frequency levels, and governors' setups. Table III shows the available governors and the sampling period (defined by the CPU vendor). Notice that the sampling period of a governor may vary by one order of magnitude depending on the CPU architecture, affecting the speed of the DVFS state update.

TABLE V
RESULTS ON A72.

	Architecture	conservative	ondemand	schedutil
1	MobileNet-V1-0.25	100.0%	100.0%	99.0%
2	MobileNet-V1-0.50	99.0%	93.8%	84.5%
3	MobileNet-V1-0.75	99.0%	100.0%	100.0%
4	MobileNet-V1-1.0	99.0%	100.0%	100.0%
5	MobileNet-V2	100.0%	100.0%	100.0%
6	MobileNet-V3-small	100.0%	94.2%	86.5%
7	MobileNet-V3-large	100.0%	100.0%	100.0%
8	EfficientNet-lite0	100.0%	100.0%	100.0%
9	EfficientNet-lite1	99.0%	100.0%	100.0%
10	EfficientNet-lite2	100.0%	100.0%	100.0%
11	EfficientNet-lite3	100.0%	100.0%	100.0%
12	EfficientNet-lite4	100.0%	100.0%	100.0%
	Macro-F1	99.7%	99.0%	97.5%
	Top-1 Accuracy	99.7%	99.0%	97.5%
	Top-2 Accuracy	100.0%	100.0%	99.8%
	Misclassifications	-	2-6	2-6

TABLE VI
RESULTS ON A57.

	Architecture	conservative	ondemand	schedutil	interactive
1	MobileNet-V1-0.25	99.0%	100.0%	100.0%	100.0%
2	MobileNet-V1-0.50	62.6%	100.0%	100.0%	100.0%
3	MobileNet-V1-0.75	93.9%	100.0%	100.0%	99.0%
4	MobileNet-V1-1.0	80.8%	59.3%	95.9%	90.7%
5	MobileNet-V2	68.0%	99.0%	99.0%	99.0%
6	MobileNet-V3-small	63.4%	100.0%	100.0%	100.0%
7	MobileNet-V3-large	77.9%	72.9%	95.1%	94.0%
8	EfficientNet-lite0	67.9%	100.0%	100.0%	99.0%
9	EfficientNet-lite1	93.2%	100.0%	100.0%	98.0%
10	EfficientNet-lite2	95.9%	100.0%	100.0%	98.0%
11	EfficientNet-lite3	100.0%	100.0%	100.0%	65.4%
12	EfficientNet-lite4	100.0%	100.0%	100.0%	65.3%
	Macro-F1	83.6%	94.3%	99.2%	92.4%
	Top-1 Accuracy	83.5%	94.5%	99.2%	92.3%
	Top-2 Accuracy	97.3%	99.7%	99.8%	99.8%
	Misclassifications	2-6, 5-8, 4-7	4-7	4-7	11-12

Table IV collects the CNNs used as benchmarks, all taken from TensorFlow HuB [22] in TFLite format; it shows the average values of latency and CPU utilization over 100 consecutive inference runs processed at the maximum frequency with 4-thread execution. The picked networks, pre-trained on the ImageNet dataset and quantized to 8-bit, belong to four distinct families: MobileNet-V1 [23], MobileNet-V2 [24], MobileNet-V3 [25], and EfficientNet [26]. The pool includes architectures with very similar characteristics. For example, networks of the MobileNet-V1 family have the same number of layers but differ in the number of filters. Furthermore, some architectures exhibit similar latency and utilization, such as Mobilenet-V1-0.50 and MobileNet-V3-small. These similarities can affect the detection accuracy of the attack. Our experiments aim to evaluate the capability of our methodology to identify the different networks, even under these challenging conditions.

B. Results

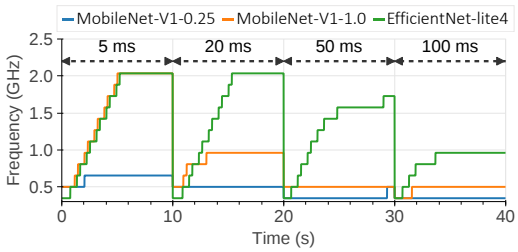
We evaluate the performance of the attacker classifier using standard metrics for multi-class classification. First, we report the F1 score, which is defined as the harmonic mean of precision and recall. Precision measures the fraction of correctly predicted positive instances (true positives) over the total number of instances predicted as positive (true positives plus false positives). Recall measures the fraction of correctly predicted positive instances over the total number of positive instances (true positives plus false negatives). Higher F1 scores indicate more accurate classifications. We also compute per-class F1 scores, which measure the performance of the model for each class individually. The per-class F1 score is computed by considering each class as positive and the remaining classes as negative, resulting in a score for each class. The average F1 score (Macro-F1) is computed by taking the arithmetic mean of the per-class F1 scores, providing an overall measure of the model's performance across all classes. Additionally, we report the Top-1 Accuracy, which is the ratio between the number of correct predictions and the total number of instances in the dataset. With the Top-2 Accuracy, we consider a prediction correct if any of the two classes with the highest prediction probabilities matches the expected label. All these metrics were evaluated on a test set having the same size as the training set, i.e. 600 traces (50 traces per network) for each CPU/governor configuration.

The experimental results presented in Tables V and VI prove the efficacy of the proposed SCA method. The reported Macro-F1 scores range from 97.5% to 99.7% for the A72 and from 83.6% to 99.2% for the A57, depending on the selected governor. These results indicate that the classifier deployed on the attacker side can accurately identify the target network, despite the different CPU architectures and the governor settings.

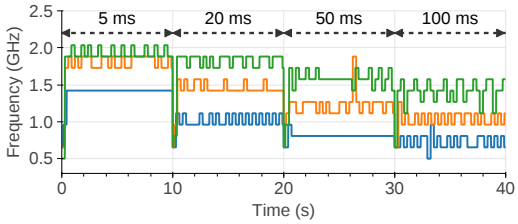
The target network and the governor sampling period primarily affect the prediction quality. Concerning the target network, we observed that most misclassifications occur between network pairs with similar characteristics, as highlighted in the last row of the results tables. This observation is consistent with the Top-2 accuracy, which reveals that even in cases where the Top-1 accuracy is low, the Top-2 accuracy is significantly higher. For instance, the A57 with a *conservative* governor has the lowest Top-1 accuracy at 83.5%, yet its Top-2 accuracy is much higher at 97.3%. Overall, the Top-2 accuracy is close to 100% in all cases, and misclassifications are limited to only a few governors and network pairs.

Regarding the sampling period, higher values lead to fewer CPU frequency switches, resulting in less information leakage in the traces. This effect is evident for the *ondemand* and the *conservative* governors, which show prediction scores on the A57 (sampling period $300\,\mathrm{ms}$) lower than that on the A72 (sampling period $10\,\mathrm{ms}$). In *conservative* mode, the Macro-F1 score is 99.7% for the A72, compared to 83.6% for the A57. For the *schedutil* governor instead, the scores on the A57 (sampling period $0.5\,\mathrm{ms}$) get higher than that recorded on the A72 (sampling period $10\,\mathrm{ms}$). Intuitively, under larger sampling periods, the probability of frequency switches reduces, and variations in CPU utilization might get missed, making it more challenging to profile the neural architecture.

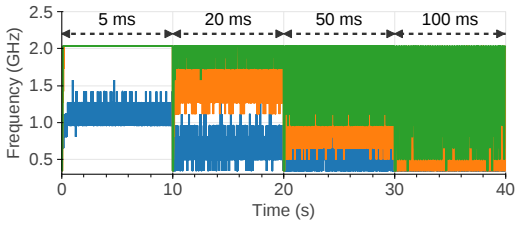
Fig. 4 provides visual evidence of the effectiveness of the proposed methodology. The figure shows frequency traces



(a) conservative



(b) ondemand



(c) schedutil

Fig. 4. Examples of frequency traces collected on the A57 with the proposed load testing methodology. The four load tests are delimited by dashed arrows, with pause lengths indicated above the arrows.

collected for three networks deployed on the A57 CPU, i.e., MobileNet-V1-0.25, MobileNet-V1-1.0, and EfficientNet-lite4, with the *conservative*, *ondemand*, and *schedutil* governors (*interactive* omitted for the sake of space). These are the same networks of Fig. 1. Unlike standard DVFS SCAs, the frequency traces are distinguishable, enabling a successful attack. The improvement is due to the testing procedure, which emphasizes the characteristics of the governors. With *conservative*, the frequency gradually increases and stops at different levels. With *ondemand*, it switches to a value proportional to the workload with small fluctuations due to system noise. With *schedutil*, it oscillates with sudden surges due to the low governor sampling period ($0.5\,\mathrm{ms}$).

Moreover, the plots demonstrate the need for multiple load tests with different pause intervals. This can be seen in the traces collected during the load test with a pause of $5\,\mathrm{ms}$ for the *conservative* and *schedutil* governors (Figs. 4a and 4c). In these cases, the traces of MobileNet-V1-1.0 and EfficientNet-lite4 do overlap, while they get distinguishable with longer pause intervals ($20\,\mathrm{ms}$, $50\,\mathrm{ms}$, and $100\,\mathrm{ms}$). We observed similar cases for other instances not reported in the figure. In summary, only a combination of multiple load tests can create distinguishable traces for all networks and governors.

V. Conclusion

This work has shown that DVFS SCAs can perform network fingerprinting in edge inference services, bypassing IaaS protection via software-based procedures. Our methodology based on load testing has proven capable of generating frequency traces that serve as signatures, enabling the detection of neural architectures with high accuracy. Our findings highlight the need for security measures to counteract such attacks, which can expose a concrete source of leakage for neural architecture stealing with severe implications for security and privacy in the context of IaaS. In particular, malicious actors with knowledge of the neural architecture can launch more efficient adversarial attacks to mislead the output prediction or white-box attacks aiming to replicate the network functionality.

References

[1] V. Peluso *et al.*, "Inference on the edge: Performance analysis of an image classification task using off-the-shelf cpus and open-source convnets," in *SNAMS*, 2019.

[2] J. Hou *et al.*, "Model protection: Real-time privacy-preserving inference service for model privacy at the edge," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 6, pp. 4270–4284, 2022.

[3] Y. Xiang *et al.*, "Open DNN box by power side-channel attack," *IEEE Trans. Circuits Syst.*, vol. 67-II, no. 11, pp. 2717–2721, 2020.

[4] X. Hu *et al.*, "Deepsniffer: A DNN model extraction framework based on learning architectural hints," in *ASPLOS*, 2020.

[5] Y. Zhu *et al.*, "Hermes attack: Steal DNN models with lossless inference accuracy," in *USENIX Security Symposium*, 2021.

[6] L. Batina *et al.*, "SCA strikes back: Reverse-engineering neural network architectures using side channels," *IEEE Des. Test*, vol. 39, no. 4, pp. 7–14, 2022.

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

[8] V. Duddu *et al.*, "Stealing neural networks via timing side channels," *CoRR*, vol. abs/1812.11720, 2018.

[9] B. A. D. Kumar *et al.*, "Inferring DNN layer-types through a hardware performance counters based side channel attack," in *AIMLSystems*, 2021.

[10] K. Patwari *et al.*, "DNN model architecture fingerprinting attack on CPU-GPU edge devices," in *EuroS&P*, 2022.

[11] Y. Liu and A. Srivastava, "GANRED: gan-based reverse engineering of dnns via cache side-channel," in *CCSW*, 2020.

[12] S. E. Arefin and A. Serwadda, "Deep neural exposure: You can run, but not hide your neural network architecture!" in *IH&MMSec*, 2021.

[13] A. Büşra and Y.-M. Ayse, "A study on power and energy measurement of nvidia jetson embedded gpus using built-in sensor," in *UBMK*, 2022.

[14] Linux Governors. Accessed on 2023/03/20. [Online]. Available: https://docs.kernel.org/admin-guide/pm/cpufreq.html

[15] D. R. Dipta and B. Gülmezoglu, "DF-SCA: dynamic frequency side channel attacks are practical," in *ACSAC*, 2022.

[16] N. Chawla *et al.*, "Application inference using machine learning based side channel analysis," in *IJCNN*, 2019.

[17] C. Liu *et al.*, "Methodology of assessing information leakage through software-accessible telemetries," in *HOST*, 2021.

[18] C. Wu *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *HPCA*, 2019.

[19] M. Taouil, A. Aljuffri, and S. Hamdioui, "Power side channel attacks: Where are we standing?" in *DTIS*, 2021.

[20] A. Kolesnikov *et al.*, "Big transfer (bit): General visual representation learning," in *ECCV*, 2020.

[21] A. Dempster, D. F. Schmidt, and G. I. Webb, "Minirocket: A very fast (almost) deterministic transform for time series classification," in *KDD*, 2021.

[22] TensorFlow Hub. Accessed on 2023/03/20. [Online]. Available: https://tfhub.dev

[23] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[24] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018.

[25] A. Howard *et al.*, "Searching for mobilenetv3," in *ICCV*, 2019.

[26] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *ICML*, 2019.