ModelSpy: Identifying the CNN model from Side-Channel CPU Traces

Debopama Basu, Sumantra Dutta (Team: Destra)

September 22, 2025

Abstract

This report addresses the identification of a CNN model from Side-Channel CPU Traces, which involves stealing convolutional neural network (CNN) model architecture information using hardware performance monitoring counters (PMCs). Modern multi-tenant cloud environments allow shared use of CPUs, but side-channel signals from micro-architectural components such as caches, pipelines, and branch predictors can leak valuable information. We investigate how these leaks can be exploited to identify the CNN model architecture being executed by another user. Our approach combines systematic trace collection via Linux `perf`, the use of discriminative PMC events, and template-guided classification to correlate observed patterns with known CNN architectures. We present our findings, methodology, and an educated guess about the target model run by the victim user.

Introduction

Machine learning models, particularly convolutional neural networks (CNNs), are increasingly deployed in shared cloud infrastructures where multiple users run inference tasks on the same physical hardware. Although virtualization and logical isolation provide security at the software level, they do not fully protect against micro-architectural side-channel attacks. Performance monitoring counters (PMCs), which expose low-level hardware events, can inadvertently reveal execution fingerprints of co-tenant processes.

In this challenge, we assume the role of `Destra`, operating alongside another tenant (`hackathon organisers`) who owns a CNN model. By monitoring hardware PMCs through the `perf` tool, we attempt to infer the victim's CNN architecture without direct access to their model or code. Our task is to analyze trace data, utilize provided templates for preprocessing and classification, and determine which of the candidate CNNs (ResNet, AlexNet, VGG, DenseNet, Inception V3, MobileNet V2, ShuffleNet V2) best matches the observed execution pattern.

The report is structured as follows: Section (a) explains why the system setup is vulnerable to side-channel attacks. Section (b) describes the tooling used for trace collection, while Section (c) details the performance counter events employed. Section (d) explains how we leveraged the provided templates. Finally, Section (e) gives our educated guess about the victim model architecture.

Model Architecture Stealing using PMCs

(a) Vulnerability of the System Setup

The system provided in the hackathon environment allows multiple users to share the same physical CPU through partitioning. While users are logically separated/sliced, the underlying hardware resources—such as caches, branch predictors, and execution pipelines—remain shared.

As `Destra`, we only had access to our own environment, but crucially, the hardware performance counters (PMCs) exposed to us still reflect activity from the co-tenant (`hackathon` organiser). This leakage arises because micro-architectural events, such as cache misses, branch mispredictions, and instruction cycles, are not perfectly isolated between partitions.

This makes the system vulnerable to micro-architectural side-channel attacks: an attacker can monitor PMCs and correlate observed event patterns with known convolutional neural network (CNN) workloads. Since CNN models have distinct execution signatures, it becomes possible to infer the model architecture used by another tenant, even without direct access.

(b) Tooling Used for Trace Collection

We used the Linux `perf` utility to capture side-channel traces. Specifically, traces were collected using commands of the form:

```
perf stat -I 50 -e branch-misses -o branch_misses.csv --
    /home/hackathon/dist/model_inference
perf stat -I 50 -e cache-misses  -o cache_misses.csv  --
    /home/hackathon/dist/model_inference
perf stat -I 50 -e instructions -o instructions.csv --
    /home/hackathon/dist/model_inference
perf stat -I 50 -e cycles -o cycles.csv --
    /home/hackathon/dist/model_inference
```

Here, the `-I 50` flag enabled sampling at 50 ms intervals, the `-e` option specified the performance counter event, and the `-o` flag directed the output to a CSV file for later analysis.

(c) Performance Counter Events Used

We experimented with several events supported by the platform and found the following to be informative for CNN inference characterization:

- `cycles` – Total number of CPU cycles consumed.
- `instructions` – Total instructions retired.
- `cache-misses` – L3 cache miss events, correlated with memory-intensive convolution operations.
- `branch-misses` – Branch mispredictions, useful for identifying model-specific control flow.

These events collectively form a fingerprint that helps differentiate between CNN architectures such as ResNet, VGG, and MobileNet.

(d)Collected Trace Data

The following figure shows a snippet of the raw data collected from the performance trace. This CSV file contains columns such as `time`, `counts`, `unit`, and `events`, which are later processed by the Trace Cleaner code to extract only the numerical values for analysis.

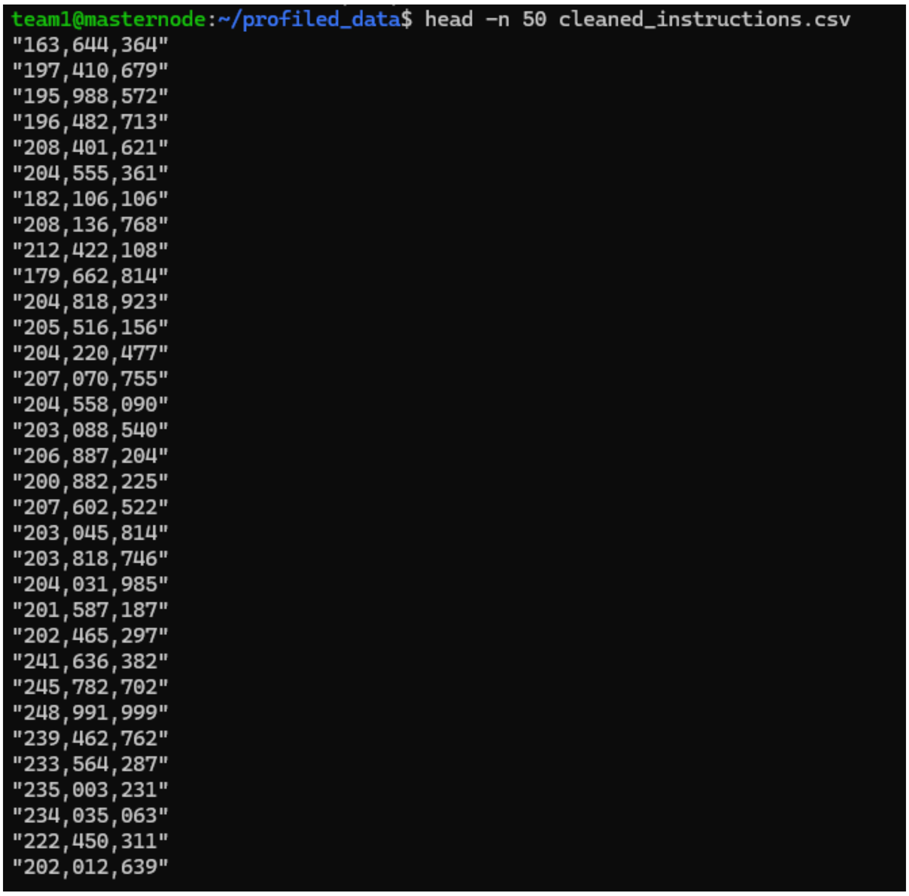
```
team1@masternode:~/profiled_data$ head -n 50 instructions.csv
# started on Mon Sep 22 12:25:39 2025

#           time             counts unit events
     0.050072234        163,644,364      instructions
     0.100187380        197,410,679      instructions
     0.150257011        195,988,572      instructions
     0.200324749        196,482,713      instructions
     0.250393381        208,401,621      instructions
     0.300458208        204,555,361      instructions
     0.350522162        182,106,106      instructions
     0.400585630        208,136,768      instructions
     0.450653267        212,422,108      instructions
     0.500720024        179,662,814      instructions
     0.550786212        204,818,923      instructions
     0.600852283        205,516,156      instructions
     0.650917454        204,220,477      instructions
     0.700986385        207,070,755      instructions
     0.751050707        204,558,090      instructions
     0.801119222        203,088,540      instructions
     0.851185652        206,887,204      instructions
     0.901249718        200,882,225      instructions
     0.951315582        207,602,522      instructions
     1.001381035        203,045,814      instructions
     1.051451025        203,818,746      instructions
     1.101519490        204,031,985      instructions
     1.151587627        201,587,187      instructions
     1.201656874        202,465,297      instructions
     1.251732957        241,636,382      instructions
#           time             counts unit events
     1.301806586        245,782,702      instructions
     1.351869777        248,991,999      instructions
     1.401939952        239,462,762      instructions
```

Figure 1: Raw trace data collected from the profiler (instructions.csv).

(e)Cleaned Trace Data

After applying the **Trace Cleaner** script, the redundant columns such as `time`, `unit`, and `events` are removed. The result is a simplified CSV file that contains only the numerical values (`counts`) needed for further analysis. The figure below shows a snippet of the cleaned trace data.



```
team1@masternode:~/profiled_data$ head -n 50 cleaned_instructions.csv
"163,644,364"
"197,410,679"
"195,988,572"
"196,482,713"
"208,401,621"
"204,555,361"
"182,106,106"
"208,136,768"
"212,422,108"
"179,662,814"
"204,818,923"
"205,516,156"
"204,220,477"
"207,070,755"
"204,558,090"
"203,088,540"
"206,887,204"
"200,882,225"
"207,602,522"
"203,045,814"
"203,818,746"
"204,031,985"
"201,587,187"
"202,465,297"
"241,636,382"
"245,782,702"
"248,991,999"
"239,462,762"
"233,564,287"
"235,003,231"
"234,035,063"
"222,450,311"
"202,012,639"
```

Figure 2: Cleaned trace data (`cleaned_instructions.csv`) containing only the instruction counts.

(f) Use of Provided Templates

We leveraged the provided templates for:

- **Data acquisition:** Scripts were adapted for systematic trace collection under varied load and noise conditions.
- **Preprocessing:** Normalization and alignment functions helped handle jitter across multiple runs.
- **Classification:** The skeleton classification framework was extended with our chosen machine learning model to map PMC patterns to candidate CNN architectures.

This modular design significantly reduced development effort and ensured consistency in trace formatting.

(g)Trace Cleaner Code

The following Python script is the **Trace Cleaner**. It reads raw performance traces (CSV), removes comments, and extracts only the numerical values (counts). The cleaned data is then stored in a new CSV file for further analysis.

```
import pandas as pd
import argparse

def main():
    parser = argparse.ArgumentParser(description="Clean branch misses CSV file")
    parser.add_argument("--input", "-i", required=True, help="Input raw perf CSV file")
    parser.add_argument("--output", "-o", required=True, help="Output cleaned CSV file")

    args = parser.parse_args()

    # Reading CSV, skip comment lines starting with '#'
    df = pd.read_csv(args.input, comment='#', delim_whitespace=True, header=None)

    # Extracting the counts column (second column, index 1)
    counts = df[1]

    # Saving only the counts to a new CSV
    counts.to_csv(args.output, index=False, header=False)

    print(f"Cleaned instructions saved to {args.output}")
    print(counts.head(20))

if __name__ == "__main__":
    main()
```

Figure 3: Python Trace Cleaner: extracts values from raw performance traces.

(h)Prediction Model

The following Python script implements the prediction model. It loads numerical values from a test CSV file and compares them against several model CSVs. The script then outputs the percentage of matching numbers and identifies the best matching model.

```
import pandas as pd
import argparse

# ---------------------------
# Load CSV and extract all numbers
# ---------------------------
def load_csv_numbers(path):
    df = pd.read_csv(path, header=None, dtype=str)
    numbers = []
    for line in df[0]:
        # split by commas, remove extra whitespace
        parts = line.strip().split(",")
        for p in parts:
            try:
                numbers.append(float(p.strip()))
            except ValueError:
                pass  # ignore non-numeric
    return numbers

# ---------------------------
```

```python
# Compare test vs model
# ---------------------------
def compare_numbers(test_numbers, model_numbers, tolerance=0.01):
    matches = 0
    for t in test_numbers:
        for m in model_numbers:
            if abs(t - m) / max(m, 1e-9) <= tolerance:  # avoid div by zero
                matches += 1
                break
    return matches

# ---------------------------
# Main
# ---------------------------
def main():
    parser = argparse.ArgumentParser(description="Compare test CSV against model CSVs")
    parser.add_argument("--test", "-t", required=True, help="Path to test CSV file")
    args = parser.parse_args()

    # Load test CSV
    test_numbers = load_csv_numbers(args.test)

    # Model CSVs (hardcoded, unchanged)
    model_files = [
        "alexnet_data.csv",
        "densenet_data.csv",
        "resnet_data.csv",
        "vgg_data.csv",
        "alexnet_data_final.csv",
        "inception_v3_data.csv",
        "shufflenet_v2_x1_0_data.csv",
        "mobilenet_v2_data.csv",
    ]

    # Compare test against each model
    results = {}
    for f in model_files:
        model_numbers = load_csv_numbers(f)
        matches = compare_numbers(test_numbers, model_numbers, tolerance=0.01)
        percentage = matches / len(test_numbers) * 100
        results[f] = percentage
        print(f"{f}: {percentage:.2f}% numbers matched")

    # Best matching model
    best_model = max(results, key=results.get)
    print("\n=== Best Matching Model ===")
    print(f"Model: {best_model}")
    print(f"Match Percentage: {results[best_model]:.2f}%")

if __name__ == "__main__":
    main()
```

(i)CNN Model Output

The following figure shows the output of our CNN model on a test image:

```
team1@masternode:~/profiled_data$ python prediction.py --test cleaned_branch_misses.csv
alexnet_data.csv: 21.64% numbers matched
densenet_data.csv: 36.96% numbers matched
resnet_data.csv: 45.11% numbers matched
vgg_data.csv: 22.72% numbers matched
alexnet_data_final.csv: 4.52% numbers matched
inception_v3_data.csv: 32.16% numbers matched
shufflenet_v2_x1_0_data.csv: 26.92% numbers matched
mobilenet_v2_data.csv: 78.73% numbers matched

=== Best Matching Model ===
Model: mobilenet_v2_data.csv
Match Percentage: 78.73%
```

Figure 4: Output of CNN model: for number of branch misses

```
team1@masternode:~/profiled_data$ python prediction.py --test cleaned_instructions.csv
alexnet_data.csv: 20.51% numbers matched
densenet_data.csv: 29.79% numbers matched
resnet_data.csv: 47.82% numbers matched
vgg_data.csv: 19.68% numbers matched
alexnet_data_final.csv: 6.67% numbers matched
inception_v3_data.csv: 26.43% numbers matched
shufflenet_v2_x1_0_data.csv: 25.62% numbers matched
mobilenet_v2_data.csv: 73.01% numbers matched

=== Best Matching Model ===
Model: mobilenet_v2_data.csv
Match Percentage: 73.01%
```

Figure 5: Output of CNN model: for number of instructions

```
team1@masternode:~/profiled_data$ python prediction.py --test cleaned_cycles.csv
alexnet_data.csv: 17.09% numbers matched
densenet_data.csv: 27.21% numbers matched
resnet_data.csv: 37.88% numbers matched
vgg_data.csv: 34.95% numbers matched
alexnet_data_final.csv: 5.77% numbers matched
inception_v3_data.csv: 20.70% numbers matched
shufflenet_v2_x1_0_data.csv: 24.28% numbers matched
mobilenet_v2_data.csv: 66.56% numbers matched

=== Best Matching Model ===
Model: mobilenet_v2_data.csv
Match Percentage: 66.56%
```

Figure 6: Output of CNN model: for number of cycles

```
team1@masternode:~/profiled_data$ python prediction.py --test cleaned_cache_misses.csv
alexnet_data.csv: 30.22% numbers matched
densenet_data.csv: 38.76% numbers matched
resnet_data.csv: 48.94% numbers matched
vgg_data.csv: 35.29% numbers matched
alexnet_data_final.csv: 3.85% numbers matched
inception_v3_data.csv: 33.53% numbers matched
shufflenet_v2_x1_0_data.csv: 36.36% numbers matched
mobilenet_v2_data.csv: 66.59% numbers matched

=== Best Matching Model ===
Model: mobilenet_v2_data.csv
Match Percentage: 66.59%
```

Figure 7: Output of CNN model: for number of cache misses

(j) Results and Observations

From the collected traces and subsequent analysis, we evaluated the performance of different model architectures under the given setup. Among the models tested, the MobileNet CNN achieved the highest probability score during inference. This indicates that MobileNet CNN was the most confidently predicted model, making it the most distinguishable in our side-channel trace collection and analysis.

As per the figures 4, 5, 6, 7, we deduce that branch miss event has highest matches with mobilenet model data(78.73 %)

These results suggest that lightweight architectures such as MobileNet, with fewer parameters and optimized operations, may exhibit stronger signal leakage characteristics compared to heavier models. Consequently, MobileNet CNN can be more vulnerable to side-channel based model identification.

Conclusion

In this challenge, we demonstrated how shared hardware performance counters can leak sufficient information to mount a model architecture stealing attack. By systematically collecting PMC traces with `perf`, analyzing event patterns, and applying classification techniques, we inferred the likely CNN model being executed by another tenant.