Compilation as a Defense

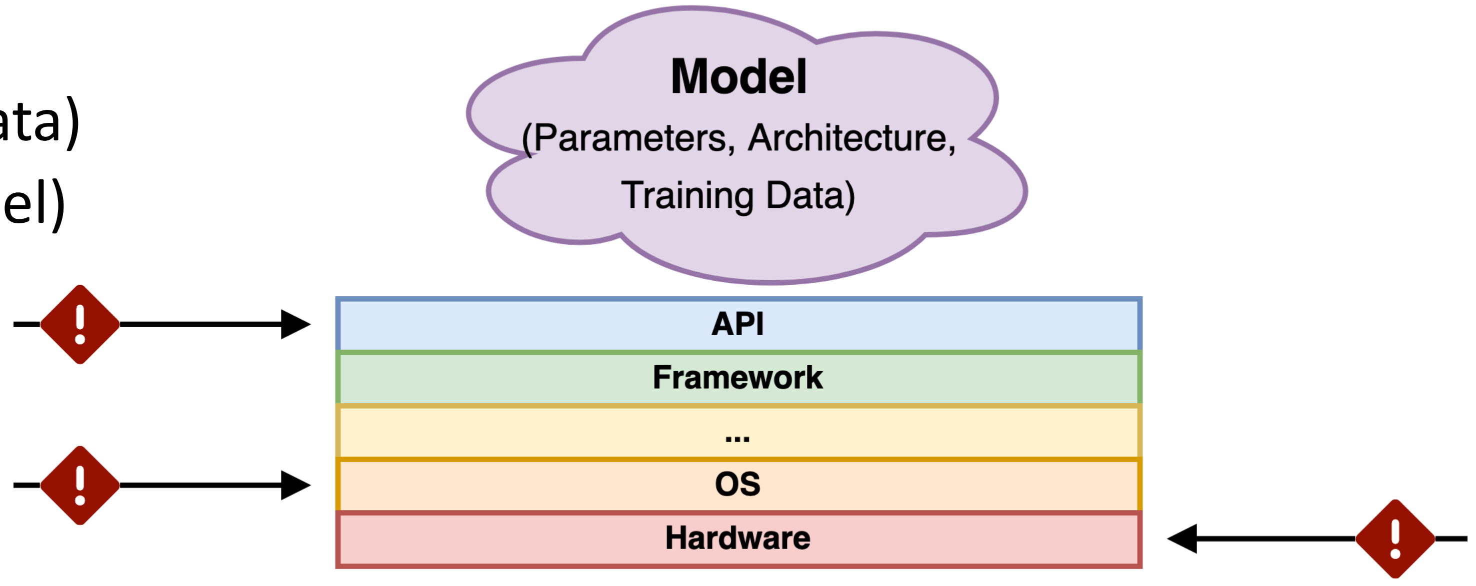
Enhancing DL Model Attack Robustness via Tensor Optimization

Stefan Trawicki, William Hackett, Lewis Birch, Neeraj Suri, Peter Garraghan

Adversarial ML (AML)

- Attacks on ML models **and** their systems

- Threat classification frameworks
  - **Extraction** (stealing)
  - **Inversion** (reproducing data)
  - **Evasion** (tricking the model)

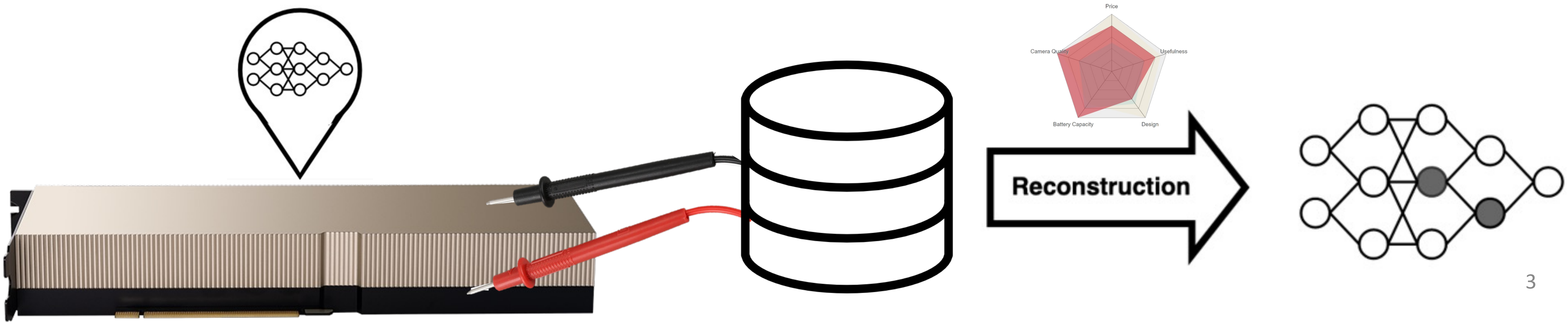


AML Side-Channel Attacks

- Extract leaky information from running processes

- Associate data with model attributes
  - Models can have a *fingerprint* left by resource access and allocation

- Extract sensitive or valuable information



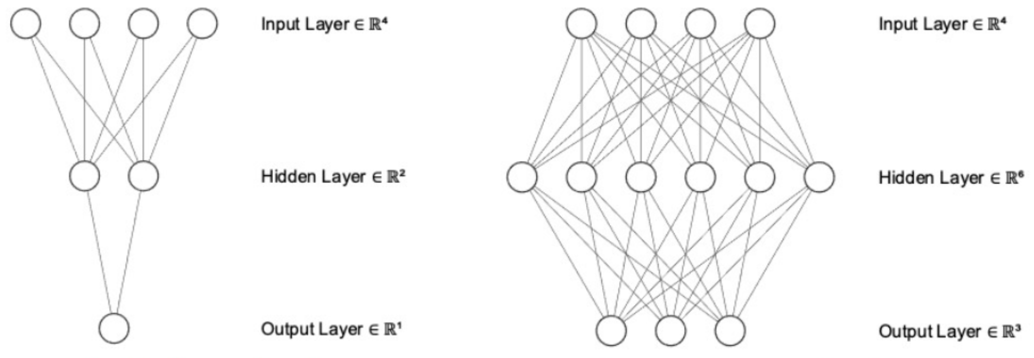
Risks Posed by Side-Channels

- Leaky information has many sources
  - Data not yet considered sensitive or important, hence unsecured
- Potentially model and dataset agnostic
- Undertaken in few inferences (< 1 second)
- Steal an architecture, parameters, data, stage further attacks

Current Defences

- Standard cybersec methods to secure system
    - But huge space to secure

- ModelObfuscator
    - Obscures and adds loop structures
    - But not model or framework agnostic
    - Model *fingerprint* can remain as before…



(a) Orginal neural nodes (b) Random neural nodes

- *A method to agnostically modify architecture and fingerprint is better…*

Objective

- Compilation as a Defence
  - Generate bespoke neural network operator implementations

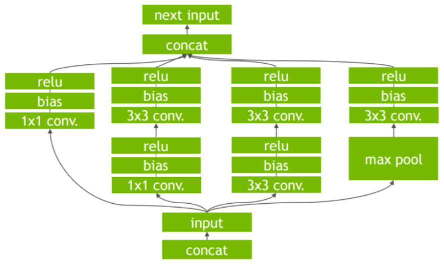
- Model operator schedule modification
  - Less readable *fingerprint* as a byproduct of optimization?
  - Break the model-process associations
  - Lower chance of reproduction

- No negative impact on inference time

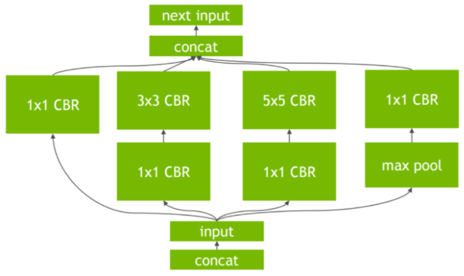
Background: ML Compilers

- Tensorflow, Pytorch, etc, provide graph representations that are mapped into executable code

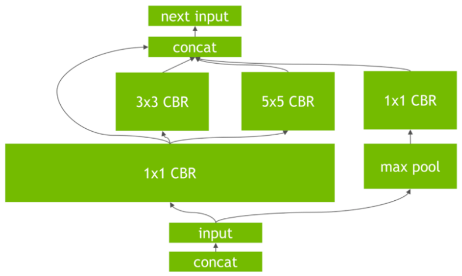
- Intermediate representations (IRs) are 'lowered'
  - Graph → tuned IRs → LLVM, NVCC → machine code
  - Lowering IRs generates unoptimized code for a machine
  - Most compilers use heuristics to apply optimizations



Original graph



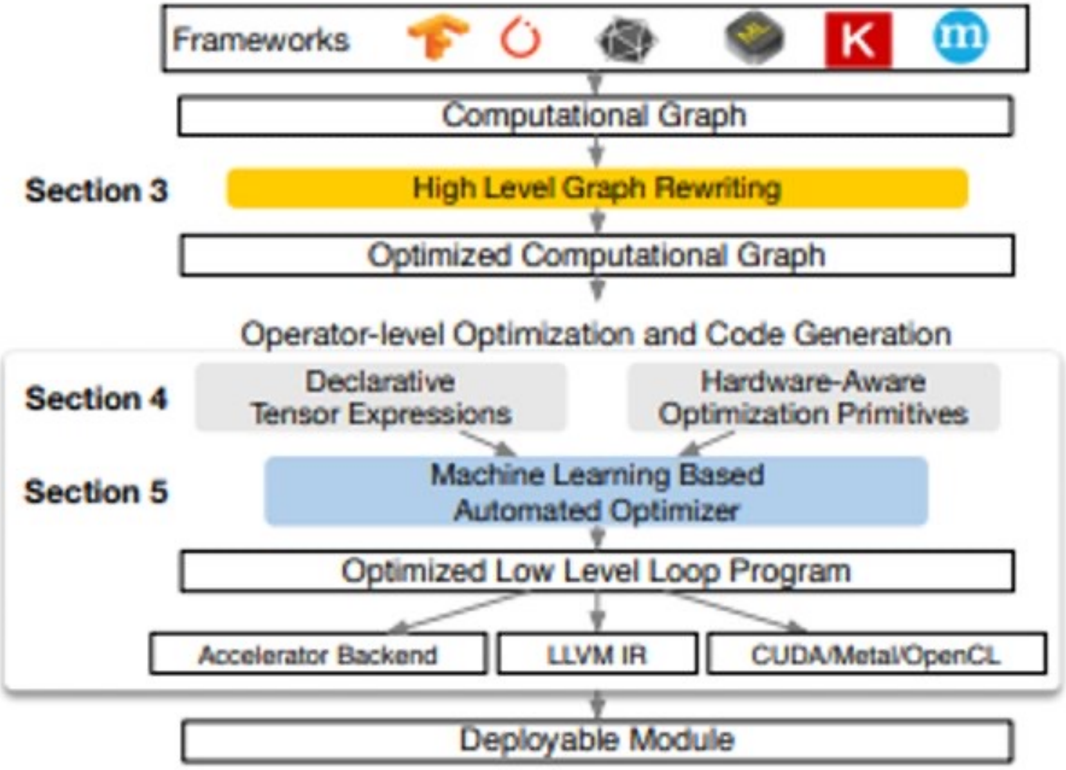
Vertical fusion



Horizontal fusion

Background: Apache TVM

- Generates bespoke implementations per machine
  - Uses simulated annealing to generate candidates
  - Runs trials guided by a tuner
- End-to-end
  - Accepts almost any frontend
  - Optimizes flow graph and operators
  - Targets almost any backend
- Model/framework agnostic
  - Leverages a very mature ecosystem



Goal

- Apply TVM to different models
  - Different domains, architectures, sizes
- Perform increasing amounts of optimization
  - More trials and better-performing tuners
- Assess whether attack success is decreased with optimized models

MINDGARD

Experiment Setup

- ResNet18, DenseNet121, RoBERTa & YoloV4
  - 8-124 million parameters
  - Multi-domain (image classifier, text, object detection)
  - All ONNX framework
- TVM parameters
  - 0 to 500 trials
  - Random and XGB rank tuner
  - Additionally, graph optimisation was tested

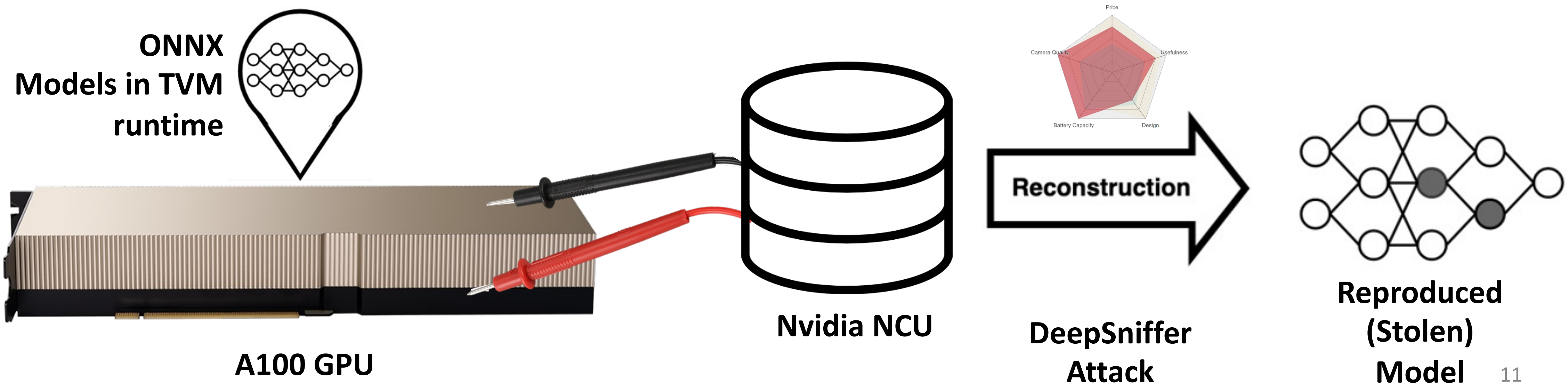
= ~240 combinations

= 83 hours of compute

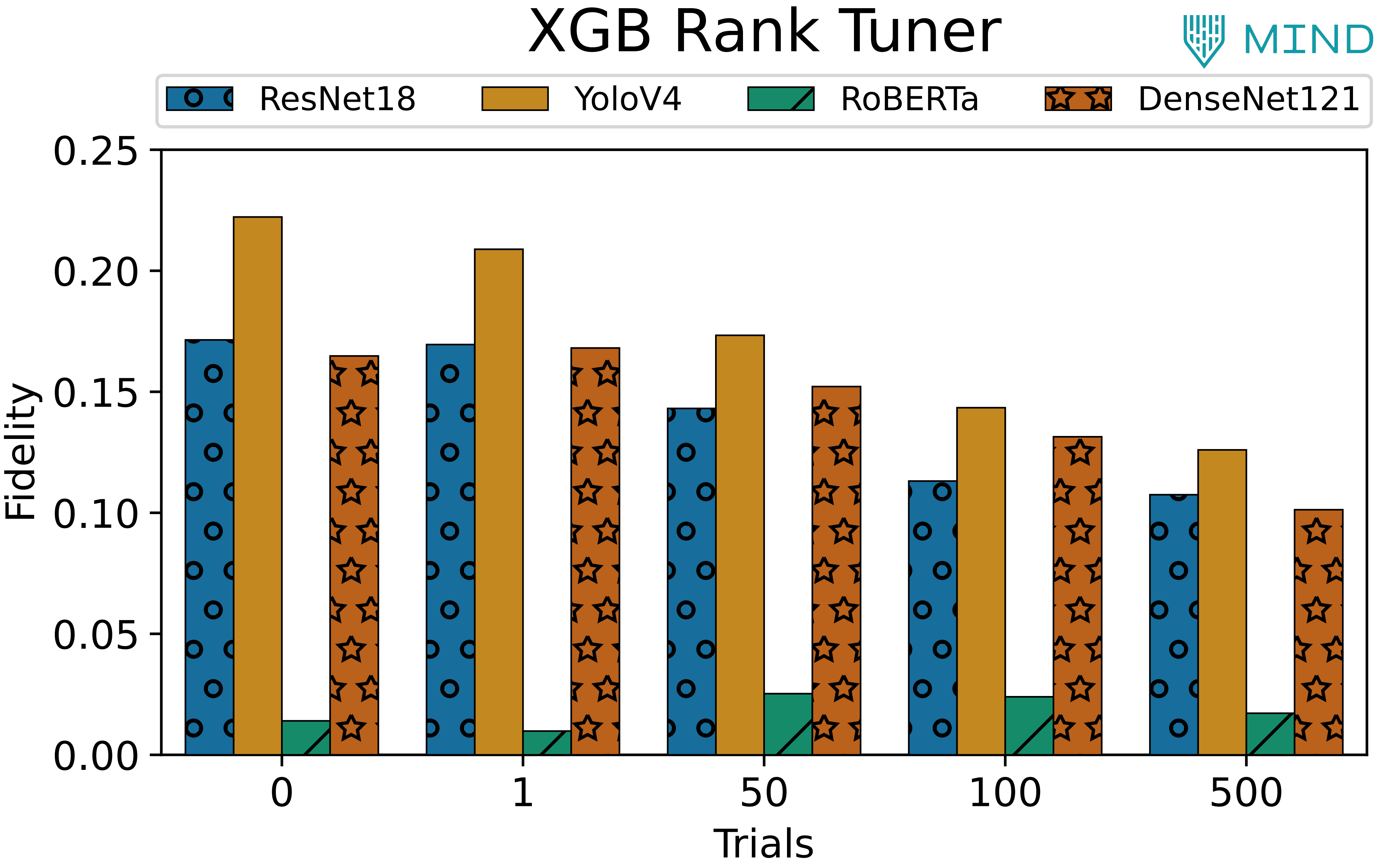
Method: Assessment pipeline

- Nvidia NCU to measure kernel memory reads/writes

- Measure reconstruction accuracy (**fidelity**) of stolen model with the **DeepSniffer Side Channel Attack**

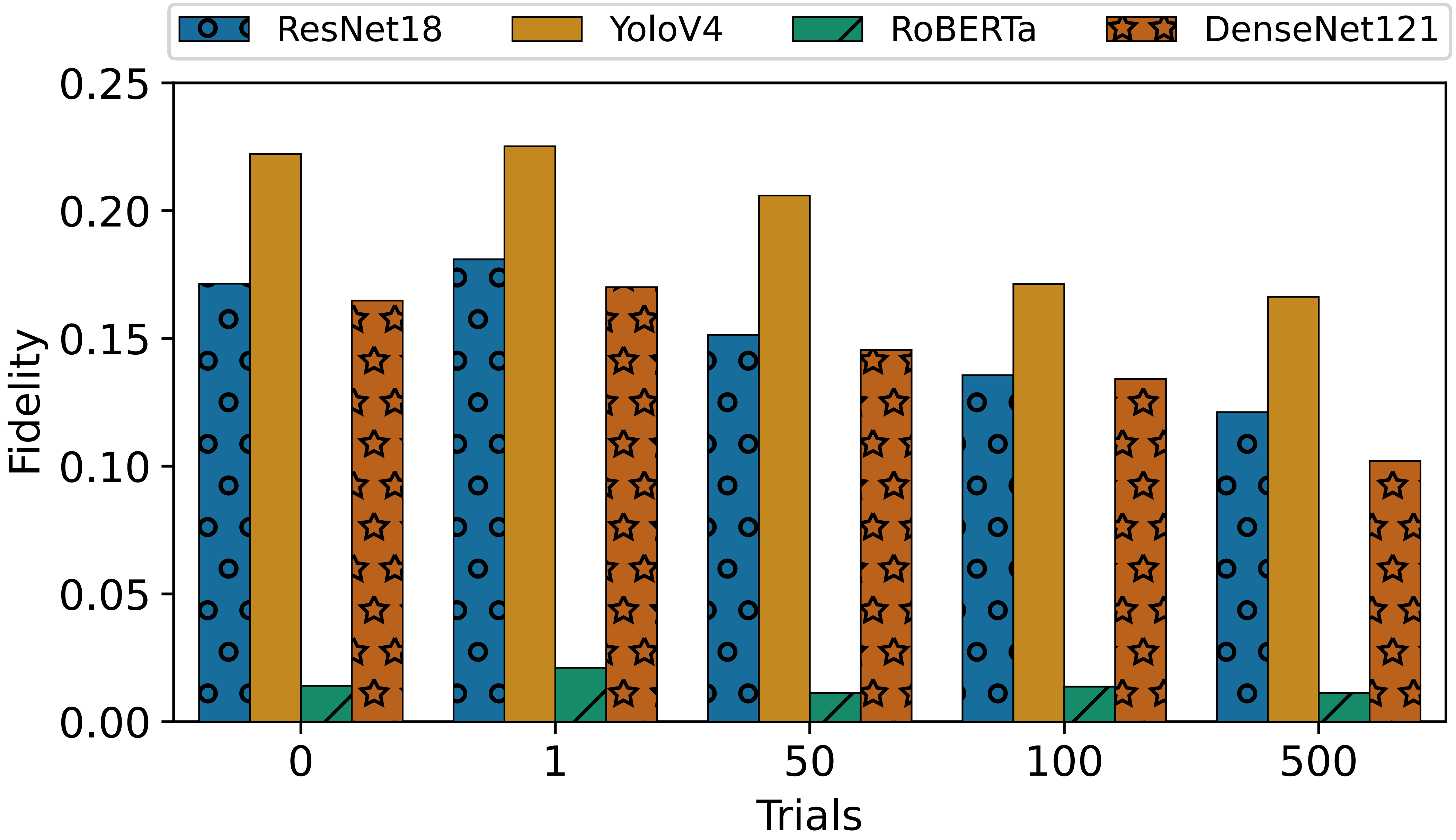


Preliminary Results


XGB Rank Tuner
ResNet18
YoloV4
RoBERTa
DenseNet121
Fidelity
0.25
0.20
0.15
0.10
0.05
0.00
0
1
50
100
500
Trials

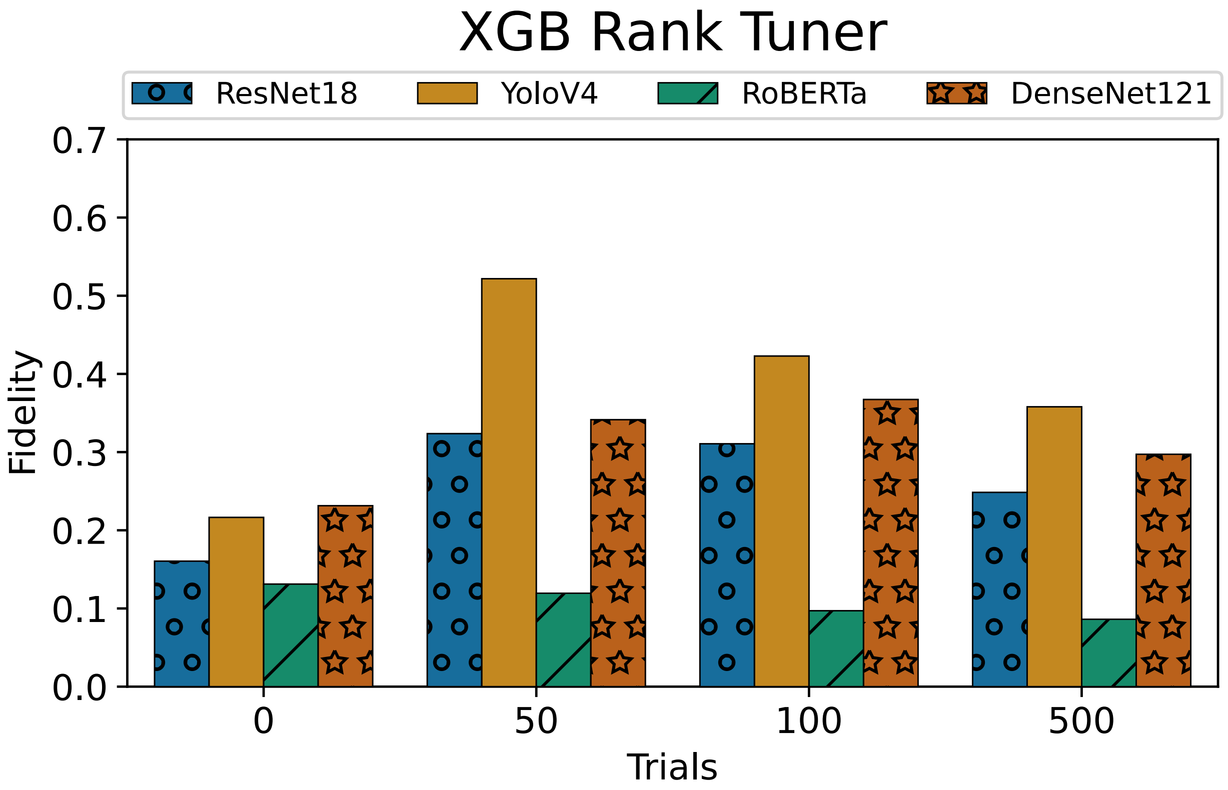
MINDGARD

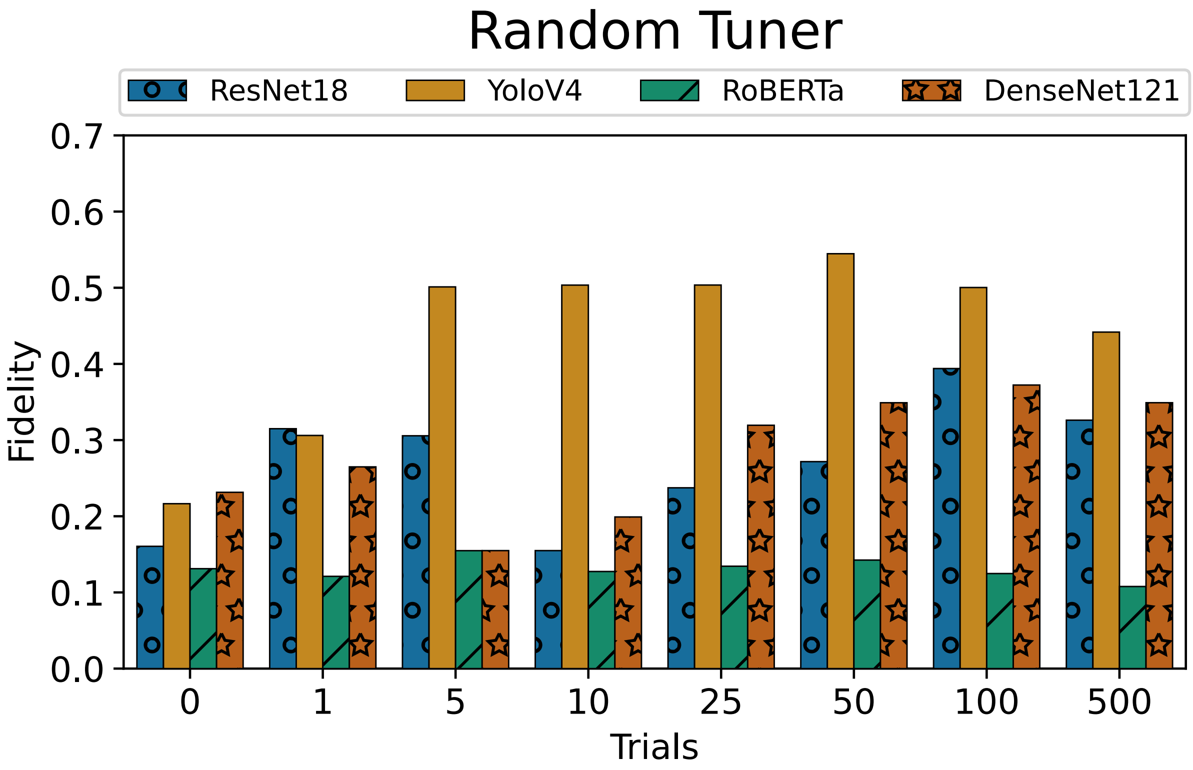
Random Tuner



Discussion

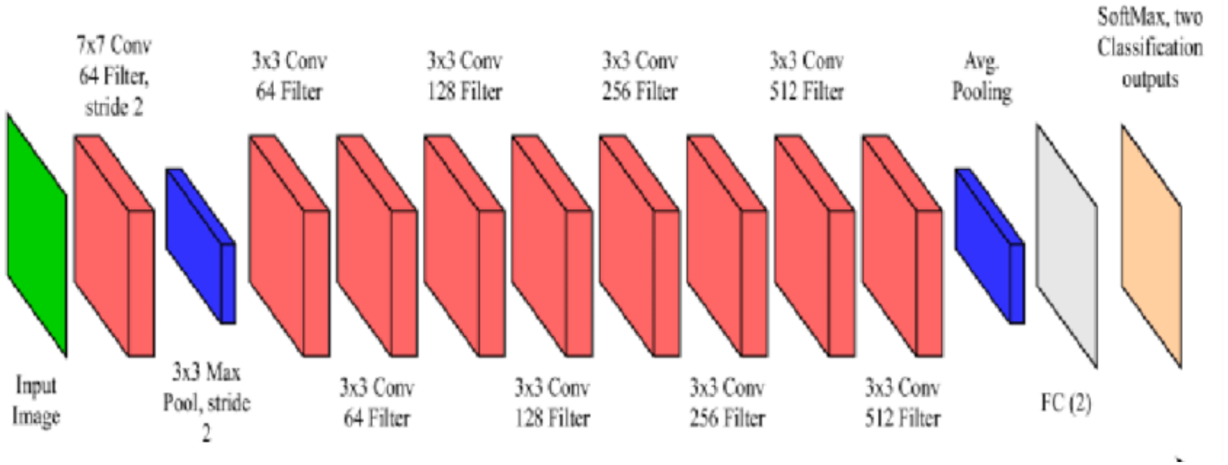
Graph Optimization

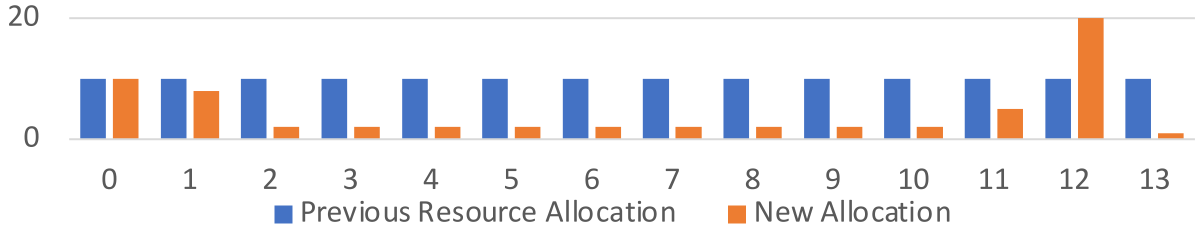




Selective Operator Optimization

- Find operators conducive to fingerprinting and optimize them heavily
  - Would require far less compute
  - Use to better guide the tuner





Utilize Ansor

- This experimentation used AutoTVM
- Ansor/Auto-Scheduler generates even more bespoke implementations

	AutoTVM Workflow	Auto-scheduler Workflow

Step 1: Write a compute definition
 	
(relatively easy part)
	`# Matrix multiply`
	
	`C = te.compute((M, N), lambda x, y:`
	`te.sum(A[x, k] * B[k, y], axis=k))`
		`# The same`

Step 2: Write a schedule template
 	
(difficult part)
	`# 20-100 lines of tricky DSL code`
	
	`# Define search space`
	`cfg.define_split("tile_x", batch, num_outputs=4)`
	`cfg.define_split("tile_y", out_dim, num_outputs=4)`
	`…`
	
	`# Apply config into the template`
	`bx, txz, tx, xi = cfg["tile_x"].apply(s, C, C.op.axis[0])`
	`by, tyz, ty, yi = cfg["tile_y"].apply(s, C, C.op.axis[1])`
	`s[C].reorder(by, bx, tyz, txz, ty, tx, yi, xi)`
	`s[CC].compute_at(s[C], tx)`
	`…`
		`# Not required`

Step 3: Run auto-tuning (automatic search)	`tuner.tune(…)`	`task.tune(…)`

MINDGARD

Other Ideas

- Frequently changing the applied optimizations
  - Moving-target

- Applying in combination with existing approaches
  - Theoretically fully compatible with ModelObfuscator

Conclusions

- Demonstrated automatic & agnostic method to increase model robustness to attack
- Attack success decreases of over 40% using tensor optimization
- Discussed avenues to expand on the preliminary work