AI Psychiatry: Forensic Investigation of Deep Learning Networks in Memory Images

David Oygenblik[1], Carter Yagemann[2], Joseph Zhang[3], Arianna Mastali[1],
Jeman Park[4*], Brendan Saltaformaggio [1*]
[1]*Georgia Institute of Technology* [2]*Ohio State University*
[3]*University of Pennsylvania* [4]*Kyung Hee University*

Abstract

Online learning is widely used in production to refine model parameters after initial deployment. This opens several vectors for covertly launching attacks against deployed models. To detect these attacks, prior work developed black-box and white-box testing methods. However, this has left a prohibitive open challenge: How is the investigator supposed to recover the model (uniquely refined on an in-the-field device) for testing in the first place. We propose a novel memory forensic technique, named AiP, that automatically recovers the unique deployment model and rehosts it in a lab environment for investigation. AiP navigates through both main memory and GPU memory spaces to recover complex ML data structures, using recovered Python objects to guide the recovery of lower-level C objects, ultimately leading to the recovery of the uniquely refined model. AiP then rehosts the model within the investigator's device, where the investigator can apply various white-box testing methodologies. We have evaluated AiP using three versions of TensorFlow and PyTorch with the CIFAR-10, LISA, and IMDB datasets. AiP recovered 30 models from main memory and GPU memory with 100% accuracy and rehosted them into a live process successfully.

1 Introduction

Law enforcement and forensic investigators need the capability to examine production deep learning (DL) models in the case of failure or attack. Existing works have proposed a slew of practical attacks against DL models, such as adversarial inputs [1]–[8], adversarial teaching [9]–[13], trust issues in federated learning [14]–[17], and attacks that backdoor the model in-memory [18], [19]. Fortunately, the research community has developed complementary vetting techniques to inspect a DL model and detect the attack [6], [20]–[26]. Ideally, an investigator would apply these vetting techniques to inspect a DL model post-failure. Unfortunately, *retrieving* the attacked DL model to inspect remains difficult for both technical and regulatory reasons.

In a forensic scenario, investigators cannot rely on obtaining a "stored copy" of the DL model for investigation. The at-rest representation of the model may be encrypted or instantiated from remote networks in countries outside the investigator's jurisdiction [27]–[29]. Assuming legal permission and vendor cooperation, prior research has explored the retrieval of DL models from compiled binaries via deep neural network (DNN) operator identification and symbolic analysis to infer the shapes, parameters, and hyperparameters of models [30], [31]. Forensic investigations rarely meet such assumptions. Further, investigators would still face significant effort to reverse engineer novel operator definitions [31] or apply symbolic analysis [30] in the face of binary protection techniques. Worse still, Sun et al. discovered that roughly 33% of ML models analyzed in their work are protected from extraction via anti-dynamic-analysis techniques [32].

DNNs may also refine their decision boundaries continuously after deployment. For example, Apple's Face ID takes pictures of the user when they correctly enter their PIN to retrain [33]. Similarly, Tesla uses fleet learning, imitation learning, and self-supervised learning to continuously update the models in end-devices [34]. Systems can also tailor themselves to a particular environment (e.g., network, user, geography) to combat concept drift [27]. Each deployed model operates with unique parameters, necessitating that any inspection be conducted on that particular deployed model, which ultimately demands the extraction of the unique on-device model.

This creates a technical dilemma for forensic investigators. They might choose to apply *black-box* analysis techniques to inspect the victim model in deployment. Unfortunately, existing literature [35] and our work (§2.1) have found black-box techniques to be slower and less accurate for attack detection. Alternatively, applying *white-box* inspection techniques (e.g., [6], [21]–[25]) requires in-memory access to the DL model. However, invasive instrumentation of the

*Co-corresponding author.

running DL model is nearly impossible. First, DNNs are implemented with complex software stacks, including interpreted languages (e.g., Python) and complicated ML frameworks whose data structures vary based on platform, version, and framework (e.g., TensorFlow [36], PyTorch [37]). Further, the white-box inspection would require visibility into the GPU, which stores weights and biases, and uses complex parallel computing platforms (e.g., CUDA [38]). Lastly, such invasive instrumentation of a victim device risks destroying evidence of the attack (e.g., crashing the ML process).

Though clearly important to the analysis of DL models, white-box techniques overlook a critical and arduous step in the investigation of the model, the actual *retrieval* of the model. In light of this, we turned our attention to memory-image forensics to recover DL models from memory, as well as enable the application of white-box inspection techniques. However, many existing state-of-the-art memory-forensics techniques rely on static data structure recovery [39]–[44], which are unable to recover the varied and complex data structures used in DL software stacks, as different models (which can be user-defined) have different in-memory representations. ML frameworks employ advanced garbage collection and memory management optimizations, obfuscating which in-memory objects are legitimate (e.g, tensors may be left allocated in memory with appropriate sizes and element counts, but with broken pointers). Layer weights are often stored in GPU memory, which has its own address space that is inaccessible to the CPU. Likewise, layer weights are stored in consecutive buffers, so it is impossible to determine where tensors start and end in GPU memory without context from CPU data structures. Lastly, to apply white-box analysis techniques, investigators must also be able to *rehost* the model into a live process for inspection.

To address these challenges, we propose AiP, an automated system for *recovering* DL models from main memory and GPU memory and *rehosting* them into a live process. AiP is the first work to enable the white-box inspection of in-production DL models by pairing GPU and CPU memory images. We come to the key insight that regardless of framework, platform, and version used in a proprietary DL system, the underlying data structures in these systems must contain key components generic to all DNNs (weights, biases, shapes, and layers such as flatten, pooling, etc.). AiP requires no prior knowledge of the particular model and can reconstruct models generically, allowing recovery of a variety of models such as CNNs [45] and RNNs [46]. AiP begins by identifying high-level DNN components associated with the model, ultimately recovering a root model object. AiP then proceeds to recover low-level data structures from CPU and GPU memory associated with the tensors of the model. AiP finally rehosts the recovered model into a new live process on the investigator's device, such that white-box techniques can be applied.

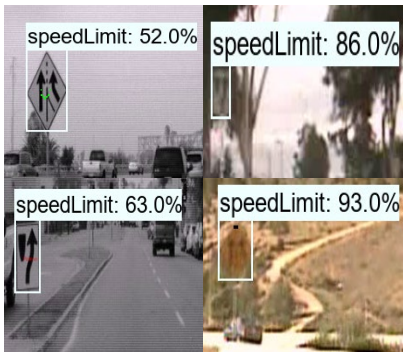

Figure 1: Backdoors used in Table 1. From the top left in clockwise order: Pattern, Pixel, Patch, and Watermark.

In our evaluation using the LISA [47] traffic sign dataset, CIFAR-10 [48] image dataset, and IMDB movie review dataset [49], AiP successfully recovered 30 different models (with upwards of 94M parameters) and achieved 100% recovery accuracy. AiP also rehosted all 30 models into the live process (equivalent to the original deployed model) to enable further analysis with white-box techniques. To facilitate future work, we have made our code and dataset open source.[1]

2 DNN Forensics with AiP

AiP is designed to be "plug and play" with any existing or future white-box investigation techniques. We aim to demonstrate why AiP is necessary and helpful for the investigator by showing (1) that AiP can cooperate with white-box investigation techniques (which would be impossible without model recovery and rehosting), (2) that the white-box introspection technique enabled by AiP is better than the black-box introspection technique, and (3) the assumptions for applying and extending AiP.

2.1 Motivating Forensic Case

Self-driving cars are edge devices with uniquely trained parameters. Proper behavior of the car's DNN is critical to the safety of the passengers in the car as well as other drivers on the road. When a self-driving car incorrectly identifies a traffic sign, that car's DNN model needs to be examined. Recall that since each self-driving car's model may be different from one another, the exact model deployed in the car at the time of misbehavior needs to be analyzed. Consider a self-driving car that has been deployed with a DL model that is continuously refined via online learning. The car has been extensively tested for backdooring *before* its deployment into the wild, but unfortunately it misidentifies a traffic sign and causes a collision.

[1] https://github.com/CyFI-Lab-Public/AiP

Table 1: Evaluation and Comparison of White-box Backdoor Detection Enabled by AiP to Black-box Detection on Models Created in TensorFlow 2.3.

BD	Model	PR[1]	ASR[2†]	Acc	DT(s)[3]		BD-C[4]		
					BB	WB	GT	BB	WB
Patch	RN152	0.00	-	97.0	-	-	∅	∅	∅
		0.06	98.4	97.2	201,389	5,198	[SL]	∅	[SL]
		0.10	99.3	97.3	204,671	5,374	[SL]	∅	[SL]
		0.16	99.4	96.7	208,653	5,221	[SL]	∅	[SL, SS]
	MN-V1	0.00	-	96.9	-	-	∅	∅	∅
		0.06	98.7	98.1	155,537	5,568	[SL]	∅	[SL]
		0.10	99.4	97.9	163,702	5,002	[SL]	∅	[SL]
		0.16	99.6	97.7	155,691	5,613	[SL]	∅	[SL]
Pattern	RN152	0.00	-	97.0	-	-	∅	∅	∅
		0.06	6.1	93.6	204,442	5,293	[SL]	∅	[SL]
		0.10	8.0	91.5	199,503	5,411	[SL]	∅	[SL]
		0.16	12.9	87.8	206,749	5,110	[SL]	∅	[SL]
	MN-V1	0.00	-	96.8	-	-	∅	∅	∅
		0.06	10.0	92.0	149,134	5,733	[SL]	∅	[SL]
		0.10	3.6	95.4	150,392	5,282	[SL]	∅	[SL]
		0.16	9.7	90.9	155,661	5,599	[SL]	∅	[SL]
Pixel	RN152	0.00	-	97.0	-	-	∅	∅	∅
		0.06	34.2	93.4	202,211	5,365	[SL]	∅	[SL, BG]
		0.10	69.6	91.5	202,983	5,244	[SL]	∅	[SL]
		0.16	66.4	91.3	209,640	5,502	[SL]	∅	[SL]
	MN-V1	0.00	-	96.8	-	-	∅	∅	∅
		0.06	9.9	96.5	161,227	5,328	[SL]	∅	[SL]
		0.10	1.3	95.9	159,130	5,346	[SL]	∅	[SL]
		0.16	7.3	93.5	153,792	5,192	[SL]	∅	[SL]
Watermark	RN152	0.00	-	97.0	-	-	∅	∅	∅
		0.06	79.2	96.3	198,818	5,096	[SL]	∅	[SL]
		0.10	81.6	95.8	200,493	5,190	[SL]	∅	[SL, BG]
		0.16	88.1	94.9	203,332	5,337	[SL]	∅	[SL]
	MN-V1	0.00	-	96.8	-	-	∅	∅	∅
		0.06	90.5	97.7	156,656	4,931	[SL]	∅	[SL]
		0.10	90.1	94.5	160,033	5,403	[SL]	∅	[SL]
		0.16	91.2	95.5	156,213	5,618	[SL]	∅	[SL, SS]

[1] Poison Rate given as one of the following: 0, 0.06, 0.1, 0.16.
[2] Attack success rate of images containing the backdoor trigger.
[3] Total detection time (DT) in seconds for Whitebox (WB) and Blackbox (BB) techniques.
[4] Backdoor detection classes:
SL: Speed Limit, SS: Stop Sign, BG: Background.

Investigators have been provided with the original model, but the DL model's decision boundaries have been (and continue to be) updated via batches of data during the online learning process. Two investigators, Alice and Bob, attempt to examine whether the car's collision was caused by an attack (i.e., implanted backdoor). Bob decides to directly apply the state-of-the-art black-box technique AEVA [50], as he cannot see the proprietary DL model running on the car. Prior to AiP, Bob's approach would have been valid because the analysis of this model was only possible through black-box techniques. However, Alice is aware of AiP and decides to first recover the DL model from the car's memory first, rehost it in a live process, and then apply a SOTA white-box technique [22]. With AiP's recovery and rehosting capabilities, we enable a white-box analysis to be applied to the unique DNN model (enabling Alice's approach).

Simulation Setup. To simulate the online learning investigation conducted by Alice and Bob, we began with unpoisoned models (Resnet152v1 and MobileNetV1), continuously fed them batches of training data, and captured memory images of the system during model refinement. To simulate an adversary slowly poisoning the car's model, we slowly poisoned new batches of images that the model's online learning trained with. The batches input to the model

(a) Backdoored street sign. (b) Zoomed sign with matching BD. (c) BD found using AiP.

Figure 2: Forensic Evidence Presented by Alice.

(in each separate experiment) contain a number of backdoored images proportional to the poison rate (0%, 6%, 10%, and 16%). The set of adversary triggers applied to the backdoored images originates from a review of previous work [14], [51], namely, patch, pattern, pixel, and watermark (Figure 1). Once each model was further refined (on the batches above) and returned to refinement on clean samples, another memory image was captured. In each scenario where Bob applies the black-box technique, we utilized AEVA [50] to test the model for any attacks. Similarly, when Alice would like to use Neural Cleanse [22] to test the model, we simulated this by recovering/rehosting the model from memory using AiP and then applying Neural Cleanse.

Simulation Findings. Table 1 shows a summary of our simulation. For each poison rate in Column 3, Column 4 shows the attack success rate (ASR) for the backdoor triggers on the rehosted models (avg of 56.4%), and Column 5 shows the overall accuracy of the models (avg of 95.4%). For the rehosted models, it is worth noting that the attack success rates (ASR, Column 4) and overall accuracy (ACC, Column 5) for each model are *exactly equivalent* in both the deployed and rehosted models. This shows that AiP's recovery and rehosting were successful (our evaluation of AiP is in §4). Columns 6-10 present the detection time and results of backdoor detection through the black-box technique (Columns 6 and 9) and the white-box technique enabled by AiP (Columns 7 and 10). For all scenarios, the ground truth backdoored class (Column 8) is speed limit (SL).

Unfortunately for Bob, the black-box technique fails to find the backdoor in all experiments. We found the maximum anomaly index for the SL class to be 2.572, falling below the baseline 4.0 needed to be met to be declared as backdoored by the technique [50]. Worse still, the average runtime is two orders of magnitude higher than the runtime for the white-box technique. Even accounting for the recovery and rehosting time of AiP (max 21,306 sec in Table 2), the black-box technique is still an order of magnitude slower.

Fortunately for Alice, in all scenarios where a trigger is present, corresponding to an attacked model, the white-box technique detects that the SL class is backdoored (100% discovery shown in Column 9). The backdoor detected by Alice using AiP is shown Figure 2c (corresponding to the [SL] backdoored class). The ground truth for this generated

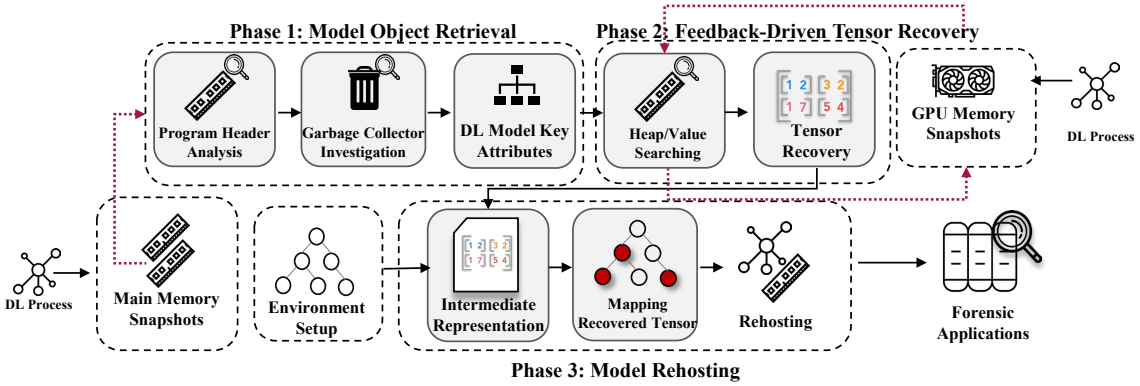

Figure 3: Overview of AiP Design and Operation.

"trigger" (patch) is shown in the middle and left images.

With AiP, the white-box analysis worked as expected. This includes finding four false positive backdoors ([SS] or [BG]) in addition to all 32 correct [SL] backdoors. Neural Cleanse [22] can produce a false positive when a mask and pattern for a non-backdoored class are generated that have an L1 Norm small enough to be anomalous. Fortunately, Alice can compare these to the much lower L1 Norm for [SL] and verify that these are false positives (as we did in this simulation). Most importantly, the white-box technique never misses the backdoored class in any of the models refined with poisoned data (0 FN).

Solving the Case. AiP provides the following evidence for Alice to solve this case:

1. Figure 2c shows the identified backdoor trigger displayed by AiP to Alice. In legal proceedings, Alice can use this output to testify that the self-driving car will misbehave 98.1% of the time this trigger is applied to signs on the street (evidenced by the 98.1% attack success rate in Table 2). For example, Alice may need to provide evidence for negligence claims, such as those brought against Tesla [52].
2. Alice can use AiP to interact with the model and demonstrate that new photographs (shown in Figure 2a) from the crash site trigger the model's backdoor. Crime scene photographs are often used to demonstrate evidence to a jury [43]. Alice can physically investigate the roads near the scene of the crash to identify candidate street signs that could have caused the crash. Figure 2b would present hard evidence to a jury that a backdoored sign present at the scene (aligning with the trigger output by AiP) could cause the accident.

Legal proceedings rely upon expert witnesses providing their interpretation of the best evidence they can obtain. Without using AiP in her forensic investigation, Alice would have no evidence that a backdoor existed and, in a court of law, would be unable to demonstrate and testify to the likelihood of the model causing the accident.

2.2 Assumptions

The design of AiP is built on several assumptions that match traditional forensic scenarios (such as the motivating example presented above). We assume that the DL model is built on commodity frameworks that are not obfuscated (e.g., TensorFlow, PyTorch). These frameworks are built on a full software stack where each software component must conform to the higher or lower software components. For example, an ML model designer creating a DNN in TensorFlow must conform the model's design to the underlying framework's classes and objects. Subsequently, the framework's classes and objects must follow very specific structures that are defined by the interpreted language and lower-level C code. Finally, lower-level data structures defined in C must conform to the complex GPU software stacks, such as CUDA [38]. These interdependencies make obfuscating or redesigning the framework code difficult, which supports our assumption that AiP will apply to most traditional forensic scenarios.

In designing AiP, we reverse-engineered this software stack and chose the lowest layers possible to target AiP's recovery. This would allow an investigator to recover a diverse set of models generically and also put AiP out of reach for all but the most advanced anti-forensics techniques. In a non-traditional forensics scenario, each layer of the software stack described above provides some opportunity for anti-forensics. A model designer attempting to obfuscate a model can do very little (simple tricks such as obfuscating model and layer names) because the model's implementation must conform to the framework. Framework designers can obfuscate classes and objects by complicating the structures of DNN nodes and intermediate data structures, but they must still comply with the lower-layer software. AiP can be extended to any new framework design (regardless of the complexity of the data structures), which we discuss §5.1. Finally, the most determined obfuscator may redevelop software at the GPU-level intending to obfuscate the ML system. This could thwart AiP, but would entail a full

redesign of the GPU interface, GPU driver, and framework, which we consider out of the scope of this paper.

3 AiP Design

Figure 3 provides an overview of AiP, which consists of three main phases. AiP takes CPU and GPU memory images as input from an investigator (detailed in Appendix A). AiP first locates the DL model's root object in memory (§3.2). AiP then applies its generic recovery algorithm (§3.1.1) to search for tensors[2] for each layer in the model. AiP then identifies whether a tensor exists in CPU or GPU memory, modifying search constraints based on where the tensor is (§3.3). AiP then connects recovered tensors to higher-level data structures, matches them to nodes in the DNN's graph, and rehosts the recovered model in a live process used for the white-box investigation (§3.4).

3.1 DNN Characteristics and Key Observations

DNNs are complex mathematical operations that can be represented as *directed acyclic graphs* or DAGs, meaning that no operation points back to previous operations. Essentially, operations will be performed from node to node in a directed fashion where no previous node is ever used again. This definition, extends beyond a particular framework or platform. This reveals the key intuition that for each DNN, which is a DAG, the semantics of the DNN's operations can be captured by examining the inputs of each node, as well as each node's shape, element counts, weights, and node type. By understanding the hierarchy of data comprising a DL model node, it is possible to interpret properly the tensors in memory that match the high-level representation of the node in the DNN's graph. The iterative application of tensor interpretation to all DNN nodes starting from the model object enables the recovery of the DNN model.

3.1.1 Tensor Recovery and Filtering

We apply the intuition that though there may be thousands of tensors in memory, each tensor recovered for the DAG should correspond to exactly one valid tensor in memory, with no ambiguity between distinct tensors with equivalent element count and shape. Starting at the DAG's rootnode, AiP employs a generic recovery algorithm, which utilizes the unique characteristics of the DNN's nodes to isolate and recover the nodes' associated tensors (framework specific implementation shown in §3.3).

AiP's generic recovery is shown in Algorithm 1. It begins with a breadth-first search through the model object for all the branches leading to leaves (tensors) (Line 3 - Line 13).

[2] A multi-dimensional array of data used in DL.

Algorithm 1: Tensor Recovery.

Input: Model Root R
Output: Tensor Set V_T

```
   // Initialize Queue, Handle List, and Constraint List
1  Q ⟵ ∅ V_handle ⟵ ∅;
2  V_cnstr ⟵ ∅;
   // Fill Constraint List for Filtering
3  Q.enqueue(R)
4  while not Q.empty() do
5      Node = Q.dequeue()
6      if Node.children = ∅ then
           // Get Handle
7          V_handle ⟵ V_handle ∪ H(Node);
           // Get Constraint
8          V_cnstr ⟵ V_cnstr ∪ C(Node);
9      end
       // Visit Children
10     for x ∈ Node.children do
11         Q.enqueue(x);
12     end
13 end
   // Begin Filtering. Initialize Mapping for Tensor Set
14 V_T ⟵ ∅
15 for t_c ∈ heap do
       // Filter Corrupt Tensors
16     if Corrupt(t_c) then
17         continue
18     end
       // Filter Semantically Invalid Tensors
19     if Name(t_c) ∉ V_handle, or Shape(t_c) ∉ V_cnstr then
20         continue
21     end
       // Add Valid Tensors to Recovery Set
22     V_T ⟵ V_T ∪ t_c
23 end
```

AiP extracts the shapes, element counts, and the names of tensors by looking at members in their data structures. AiP then extracts a handle to a representation of each tensor containing an internal name, data type, and device placement string (Line 7). AiP constructs a set of valid tensors, matching the constraints of the data structures created previously. The constraints created from the universal characteristics of each DNN node help AiP avoid tensors with invalid shapes or element counts, recovering only correct nodes from the DNN's graph (Line 21).

AiP also ensures that, given a tensor constraint (shape, element count, name), all pointers in the data structures recovered point to valid regions of memory in process. Pointers are checked such that they point to subsequent valid data structures (e.g., a tensor points to a buffer object with a flat in-memory buffer). AiP performs a final sanity check to find whether a tensor belongs to a node in the DNN's graph by validating the node's reference count (a count of 0 mean the tensor has been deallocated but has not been overwritten in memory).

3.2 Model Object Retrieval and Memory Forensics Frontends

Given an input of CPU and GPU memory images, AiP recovers the DNN's graph, as well as the graph's nodes, discussed in §3.1.1, to enable DNN investigation with white-box techniques. To this end, AiP first must identify a high-level object in memory fitting the characteristics of a DNN. Then, AiP recovers the root objects from the memory associated with the DNN model and its backbone. However, without an implemented memory forensics frontend for a particular system and framework, AiP would not be able to locate the nodes and graph structure that form the DNN in the tensor recovery step.

The memory forensics frontends that AiP requires to operate, all share common characteristics. These characteristics are similar from framework to framework, as the efficient management of matrices and matrix operations are standardized by common libraries such as CUDA [38], cuBLAS, and BLAS [53]. The nodes associated with the DNN need to be structured such that these standardized libraries used across platforms and systems can utilize them. As a result, all memory forensics frontends utilized by AiP share universal characteristics, as shown in §3.1. DNN model management by frameworks typically needs the following: DNN layer counts, layer names, tensor shapes, device location strings, and pointers to tensor data structures containing tensor weights. The objects leading to these characteristic DNN components may ultimately vary from framework to framework, begging the need for a memory forensics frontend. AiP, once given this memory forensics frontend, can accurately reconstruct the graph structure and node types for a DNN (as well as associated tensor weights) by following the generic recovery methodology in §3.1.1.

Python Memory Forensics Frontends. AiP comes prebuilt with memory forensics frontends for Python and the TensorFlow [36] and Pytorch [37] frameworks. Tensorflow's implementation is different from that of Pytorch's, but as discussed above, the underlying representation of tensors is largely similar. Also, the memory forensics frontend component that handles the Python interpreter is the same between frameworks. As a result, there are largely shared components of each frontend. Once given the frontend for each framework, AiP traverses the garbage collector (GC) of the Python process to recover objects corresponding to a high-level representation of the DNN model, leaving lower-level (C, C++) objects for later recoveries, such as model nodes and tensor weights.

To do this, AiP locates the Python process in memory by walking the pointers in the linked list in kernel memory containing all active processes. Using the pointer to the section header table from the ELF header, the location of the dynamic symbol table (pointing to relevant process-specific functions and symbols) is used to recover symbols necessary

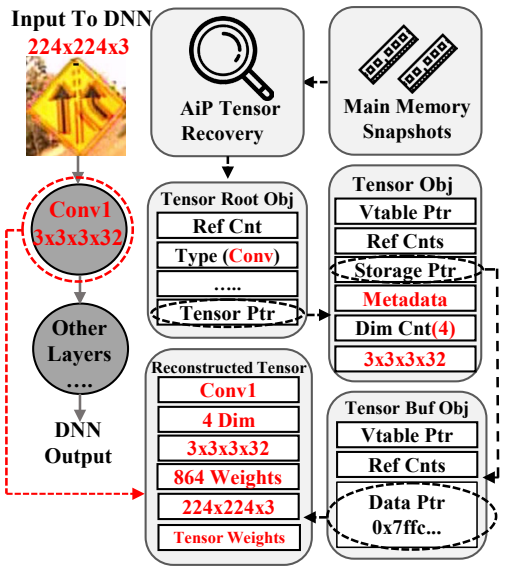

Figure 4: MobileNetV1 with SSD applied to an input image. Its first convolutional layer is shown to be reconstructed by AiP's tensor recovery (Phase 2).

for model object identification. For the Python frontend, AiP finds the Relative Value Address (RVA) of `_PyRuntime`, which is an object that points to the GC for many versions of Python, and begins looking for the model object. More memory forensics frontends can be created for other processes/interpreters such that they can be extended to other languages and frameworks.

Garbage Collector and Model Signature Search. For the prebuilt example, by finding the GC of the Python interpreter, AiP finds references to Python data structures that lead to the definition of the DNN. AiP then iterates through the GC and enumerates all Python objects referenced by the GC, searching for objects matching those of the DAG described in §3.1. AiP uses high-level common characteristics of ML models to match generic model objects to the Python objects found by traversing the GC, leading to the identification of the user-defined model. These model objects can be identified solely by name (often by a type string such as 'ResNet'), but then are verified by looking at lower-level components of that object. As data structures, these model root objects can be envisioned as a tree, containing a root node and branches leading down various Python data structures, with leaves containing the individual operations used by the graph.

Referring back to §2.1, Alice recovers an SSD MobileNetV1 model from a memory image (deployed with TensorFlow) for white-box investigation. Figure 4 shows the SSD MobileNetV1 DNN with a convolutional layer, intermediate layers (comprising the rest of the model), and the DNN output. Given an input traffic sign image (size **224x224x3**), operations will be applied such that the output of the DNN is the sign's class (e.g., speed limit).

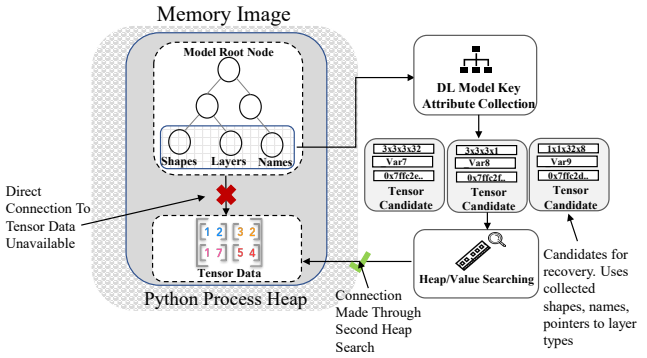

Figure 5: Generic tensor recovery allows AiP to go from high-level node characteristics to its underlying tensor representation.

We can see that the first layer (convolutional layer circled in red) is defined hierarchically in memory. First, the layer node has an associated object (**Tensor Root Obj**) that contains the object's type (Conv), ref count, and a pointer to another object, (**Tensor Obj**). **Tensor Obj** defines more characteristics of the node, such as reference counts for the layer, shape data, and dimension count. It also includes a pointer to tensor weights (**Tensor Buf Obj**). **Tensor Buf Obj** finally contains a pointer to raw tensor data stored on the GPU. By understanding the hierarchy of data shown, which we implemented as the TensorFlow frontend, it is possible to interpret properly tensors in memory (**Reconstructed Tensor**) that match the high-level representation of the node in the DNN's graph. Once these nodes are identified, the iterative application of tensor reconstruction to all DNN nodes enables the recovery of the DNN model.

Extensibility. As described in §3.1, frameworks across platforms and systems have similar representations of tensors. This is due to the way DNNs are represented, primarily as DAGs, meaning there is a list of features that each node in the DNN must contain. Therefore, memory forensics frontends for a framework need to be implemented only to the point where a model root node can be identified (discussed in §5.1) and then AiP's generic recovery can be applied.

3.3 Feedback-Driven Tensor Recovery

For model objects in memory, direct connections to tensor buffers for each operation in the model may be unavailable (via direct pointer traversal). Unless explicitly requested, large tensor buffers (tensor data) may not be directly stored in the Python layer, as shown in Figure 5. We see with solid connecting lines that pointers from the model root object can be used to identify a layer by name and shape. However, the associated tensor data (holding weights) is not accessible via pointer traversal from the model root layer (red cross). Thus, AiP identifies connections that are missing from the model graph, creates signatures for tensor objects based on collected Python layer attributes (DL Model Key Attribute Collection), and utilizes those signatures to recover tensors from the C layer, utilizing the recovery strategy in §3.1.1. Referring back to the poisoned MobileNetV1 with SSD (one TensorFlow deployed model investigated by Alice in §2.1), we see that three tensor candidates were found for the first layer of the model. However, AiP is able to discard all tensor candidates except the one precisely matching that of one candidate (Var7 with shape **3x3x3x32**).

Multi-stage Model Recovery. During tensor recovery, AiP creates the connections missing between nodes and their associated tensor buffers. To do this, AiP employs the generic multi-stage retrieval process shown in §3.1.1. During this procedure, AiP gathers handles to C tensor identifiers containing information such as device placement, internal name, and type name. These identifiers do not directly contain the weights of the tensor but hold information necessary for the ML framework to later access the weight buffer of the tensor (as indicated by the invalid connection from the model to tensor data in Figure 5). Tensor buffers might not even be available in CPU memory, as ML systems typically utilize powerful GPUs for inference and training. Fortunately, utilizing the identifiers for each tensor, AiP interprets each tensor's buffer pointer such that the buffer can be recovered from the GPU (§3.3.1).

3.3.1 Recovery of Models on the GPU

In production ML systems, GPUs are often used to perform calculations instead of a CPU. As a result, AiP should not limit its recovery to systems running on a CPU and main memories (called CPU memory for convenience). The GPU memory space is partitioned into multiple regions such as global memory, shared memory, texture memory, etc. Global memory is a region of memory on the GPU where buffers of data are created and stored. As AiP is primarily concerned with the recovery of the buffers of tensors, AiP must be made aware that the recovery of tensors is not limited to CPU memory but also the global memory associated with the ML process. However, AiP initially is unaware that pointers in CPU memory may point to regions of memory on the GPU. Without knowledge that data may be placed on the GPU (and how), AiP is unable to recover tensor weights.

However, two main technical challenges impede GPU tensor recovery. First, the determination of how data is stored on a GPU, as well as the layout of data structures on the GPU, is unclear. Second, we found that ML systems handle the management of higher-level data structures exclusively on the CPU and delegate the GPU to hold buffers of raw floats, meaning there are no meaningful data structures to search for on a GPU memory image (for tensors).

GPU Address Contextualization. To address the first challenge, we observed that the only relevant section of memory that needed to be examined was the global memory

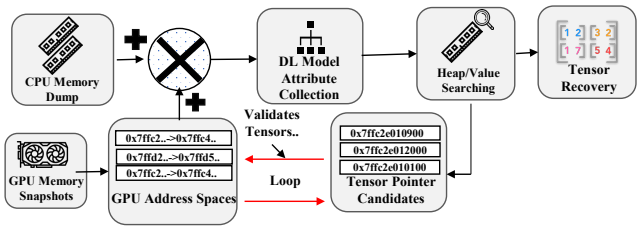

Figure 6: Feedback mechanism validating tensors.

on the GPU memory image. Regardless of how many sections of global memory are associated with the given process, we found that all buffers containing the floats corresponding to each node of the DNN are contained in these sections. Thus, it is not important for AiP to understand the semantics of the GPU memory layout to guide its recovery as long as the regions of memory associated with the tensor buffers are identified. To do this, AiP first recognizes from the CPU memory image whether the GPU memory image will even be used (looking at device location strings from tensors collected in §3.1.1). Then, AiP iterates through GPU memory regions and determines the valid address spaces to which pointers in CPU memory may point. To do so, AiP identifies all sections of global memory in GPU memory and discards all other sections as potential candidates for tensor locations. The number of relevant address spaces (automatically found by AiP) depends on the way each framework uses CUDA.

To address the second challenge, we found that the management of pointers to the float buffers is handled internally by the ML framework on the CPU, as ML frameworks do not mandate that models or tensors be stored on the GPU, instead allowing users to delegate where the tensors/model are stored. From this we see that if AiP understands that pointers in CPU memory may point to GPU memory, each buffer of floats in global memory can be interpreted. To that end, AiP can connect the node in the DNN's graph to its tensor weights via the pointer to global memory found in the previous steps of tensor recovery (information collected in §3.1.1). Though each framework manages the handling of CUDA contexts differently, when AiP recovers the node characteristics necessary to reconstruct the DNN's graph structure, AiP can recover all remaining missing tensor data from the GPU global memory. As a reminder, in §3.1.1, AiP collects device information from all tensors. For GPU tensor recovery, AiP then updates its search criteria for tensors (updating the address spaces that AiP allows in its recovery) once it is confirmed that a tensor is stored in the GPU memory (as shown in Figure 6). As seen in Figure 6, the candidates for tensor pointers are checked against valid global memory regions from GPU memory. Tensors whose data pointers do not point to addresses within the process heap are not necessarily invalid, instead pointing to valid locations within GPU global memory. However, with the knowledge that invalid tensors may be present in AiP's

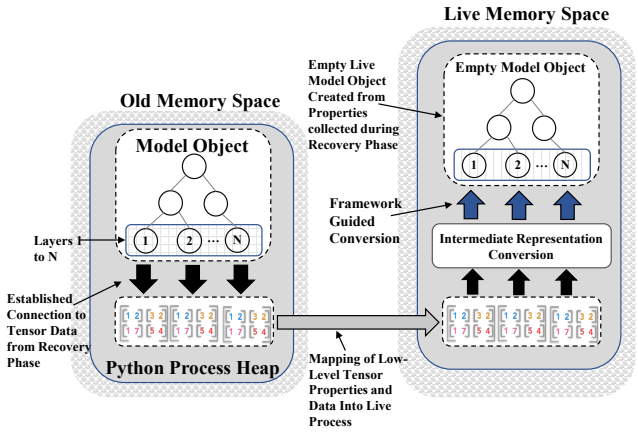

Figure 7: AiP's rehosting process. AiP utilizes an intermediate conversion and ML framework-specific tools to rehost recovered tensors.

tensor recovery, AiP uses feedback from the GPU memory image (sections, address spaces, etc) to cut invalid tensors from its recovery. In Figure 6, we see from the heap value search three tensor data pointer candidates were found for the first Convolutional layer of the model (referring back to Alice's investigation of the poisoned SSD MobileNetV1). However, as discovered previously, for each node in the DNN's graph, there should be exactly one valid pointer to its data. AiP cross-checks each data pointer with the GPU memory (with node characteristics collected earlier) and eliminates two of the three as candidates, as they did not point to GPU address spaces in global memory and their nodes did not have non-zero reference counts. By checking whether a set of floats is occupying the space in GPU memory where the tensor points to, AiP validates tensors as GPU tensors and continues recovering the raw floats necessary for DNN reconstruction.

3.4 Rehosting Recovered Models

With the recovered DNN model, AiP next *rehosts* it in a new live process to enable investigators to apply white-box investigative techniques. Although all tensors of the DNN, as well as the DNN's topology, are recovered, they are bound to the process from which they were recovered (in the memory dump). Operations applied to tensors (convolutions, flattening, etc.) access the underlying C++ tensor data structure, which contains pointers to subsequent buffer data structures, shapes, and flat in-memory data buffers. To use the DNN in a live process, recovered data structures need to first be *rehosted* within a new running process on the investigator's system. To this end, AiP maps the heap of the original process containing the recovered data structures into the live memory of a new process (as well as the GPU memory). Then, AiP reinterprets the data structures to graft them into the appropriate places within the live model.

Environment Setup. To ensure that live tensors can be replaced with tensors recovered from memory, AiP first maps the recovered tensors into the memory of a running process. This mapping contains connections within the recovered data structures that are invalid in the live process. Initially, mapped data contains pointers that cannot be followed to data in the live process, making them unusable. However, AiP uses data structure definitions for an ML framework to reinterpret these invalid data structures in a way such that they can be used in the live process with **valid** connections. Using the high-level DNN object description collected in §3.2, an empty placeholder model can be created in the live process with the same graph structure as that in the old memory image. AiP then applies the process shown in Figure 7 to reconnect tensors back to the leaves of this empty model.

Rehosting. Rehosting relevant data structures from a DNN in a memory image into a live process is challenging for three reasons. First, the locations of objects (e.g, tensors) and functions in the memory of a running process are randomized with each execution of the process as a result of Address Space Layout Randomization (ASLR). With each memory snapshot taken of a DNN, the pointers to tensor data structures and their dynamic function tables will vary. Second, tensor data structures are polymorphic and are utilized by higher-level data structures. Without handling by AiP, recovered data structures are unusable within the live process since the interconnection between higher-level data structures and recovered tensors do not exist. Finally, tensor data is typically stored on the GPU, meaning that pointers pointing to the floats for each tensor will not point to locations within the CPU memory image (making them unusable). Without mending the broken pointers between the data in the CPU and GPU memory images, the corresponding data structures, and dynamic function tables in the live environment, further analysis of the DNN with any white-box methodology is impossible.

To overcome these challenges, we realize that ML frameworks, such as PyTorch and TensorFlow, allow for the conversion of tensors represented with other popular libraries (e.g., Numpy) into framework-specific containers, such as *tensorflow.Tensor* [54] or *torch.tensor* [55]. However, the memory forensics frontends built for Pytorch and TensorFlow are not concerned with this, as, though currently unusable in a live process, the DNN's topology, nodes, and node characteristics have correctly been recovered by AiP. As seen in Figure 7, AiP first maps low-level tensor data from the old memory space into the live process. However, without fixing broken pointers to tensor data, their use in the live process is impossible. To overcome this, with the intent to make AiP as ML-framework agnostic as possible, AiP utilizes common representations of tensors shared between ML frameworks. AiP rehosts tensors to an intermediate form (a NumPy array), fitting a representation of typical tensor data structures across Python-based frameworks. When tensors are first rehosted in a common intermediate form that multiple frameworks can utilize, AiP is no longer overly dependent on the framework and framework version used by the system in which the memory dump was collected. AiP follows this ML framework-agnostic principle, allowing the ML framework to do the heavy lifting in converting these intermediate form tensors into framework-specific usable live data structures (making them usable in a live process).

For its rehosting, AiP first reloads the node constraints for model recovery such as shape, and element count (§3.3). AiP traverses the pointers in the recovered data structures mapped into memory (which are only valid in old memory) and validates the shape and element count for each tensor. AiP ensures that they are correct by comparing the number of floats recovered for each layer to the number of expected floats for each layer. AiP then locates the start of the in-memory flat buffer holding the weights of the tensor, which may be in CPU or GPU memory.

AiP then creates empty intermediate tensor representations, such as a NumPy array, with shapes corresponding to those from the node characteristics. AiP fills these arrays with the data collected from each in-memory buffer, preparing the tensors to be reused in the live process. This process avoids utilizing management objects (objects between the model root and tensor), simplifying the rehosting process. Once converted to NumPy arrays, AiP can use ML framework-specific capabilities (NumPy to tensor conversion) to convert these intermediate representations of the tensors into a form that can be plugged into a model layer (as indicated by the blue arrows in Figure 7). This avoids the direct use of third-party libraries (minimizing dependency on tools like ONNX [56]), but allows for later downstream usage if desired by the investigator. In fact, tools such as ONNX require white-box access to the model to even be usable. AiP is able to rehost tensors from the GPU memory onto the CPU memory (and vice-versa), as each ML framework manages the placement of tensors onto specific devices during conversion (with a device string). Each intermediate representation is converted to a usable tensor in the live process, and framework-specific functions are used by AiP to connect the recovered underlying tensor weights to the nodes of the DNN such that the DNN can subsequently be analyzed.

4 Evaluation

We developed the prototype of AiP for the two most popular DL frameworks, TensorFlow (TF) [36] and PyTorch (PT) [37], and show AiP's evaluation in this section. AiP's techniques can be extended to other frameworks and programming languages (detailed in §5.1).

Table 2: AiP Model Recovery from Memory Images.

FW	Ver.	Model	Layers (#)	Layers (%)	Weights (#)	Weights (%)	GPU Ptrs.	Objects Python	Objects C	Dens.[1]	Inv Tens.[2]	MGO[3]	Time(s)
TensorFlow	2.2	Resnet152v1	522	100.0	94,583,573	100.0	940	355,033	9,276	0.58	674	5,375	21,306.9
		MobileNetV1	94	100.0	21,378,133	100.0	145	356,010	1,339	0.64	81	807	13,542.7
		MobileNetV2	161	100.0	6,487,724	100.0	268	289,896	2,508	0.63	158	1,499	3,679.8
		VGG16	25	100.0	16,456,108	100.0	34	300,105	314	0.64	19	190	10,771.5
		BD-LSTM	7	100.0	2,757,761	100.0	14	286,152	137	0.65	8	42	10,089.2
	2.3	Resnet152v1	522	100.0	94,583,573	100.0	940	342,507	8,720	0.63	535	2,416	9,550.5
		MobileNetV1	94	100.0	21,378,133	100.0	145	335,444	1,507	0.54	123	414	2,132.4
		MobileNetV2	161	100.0	6,487,724	100.0	268	339,320	2,322	0.53	188	1,183	12,064.6
		VGG16	25	100.0	16,456,108	100.0	34	287,459	382	0.49	36	105	14,884.1
		BD-LSTM	7	100.0	2,757,761	100.0	14	276,576	137	0.65	8	39	9,780.8
	2.4	Resnet152v1	522	100.0	94,583,573	100.0	940	390,855	9,976	0.52	849	2,730	18,875.3
		MobileNetV1	94	100.0	21,378,133	100.0	145	386,335	1,647	0.48	158	449	12,923.1
		MobileNetV2	161	100.0	6,487,724	100.0	268	330,571	2,940	0.50	266	803	13,359.0
		VGG16	25	100.0	16,456,108	100.0	34	337,607	370	0.51	33	102	1,046.1
		BD-LSTM	7	100.0	2,757,761	100.0	14	321,042	169	0.48	16	47	9,755.8
PyTorch	1.6	Resnet152v1	364	100.0	60,344,387	100.0	777	191,761	1,864	1.0	0	1,609	460.3
		MobileNetV1	83	100.0	3,232,991	100.0	137	2,460	328	1.0	0	304	91.8
		MobileNetV2	146	100.0	6,487,776	100.0	268	188,789	640	1.0	0	582	358.0
		VGG16	38	100.0	16,456,108	100.0	34	4,062	68	1.0	0	123	74.4
		LSTM	5	100.0	920,385	100.0	11	9,694	22	1.0	0	30	122.2
	1.10	Resnet152v1	364	100.0	60,344,387	100.0	777	237,601	1,864	1.0	0	2,694	532.6
		MobileNetV1	83	100.0	3,232,991	100.0	137	5,230	328	1.0	0	493	101.3
		MobileNetV2	146	100.0	6,487,776	100.0	268	235,059	640	1.0	0	946	443.3
		VGG16	38	100.0	16,456,108	100.0	34	5,734	68	1.0	0	123	94.8
		LSTM	5	100.0	920,385	100.0	11	8,316	22	1.0	0	30	361.9
	1.11	Resnet152v1	364	100.0	60,344,387	100.0	777	237,845	1,864	1.0	0	2,694	648.0
		MobileNetV1	83	100.0	3,232,991	100.0	137	235,748	328	1.0	0	493	91.8
		MobileNetV2	146	100.0	6,487,776	100.0	268	239,455	640	1.0	0	946	448.6
		VGG16	38	100.0	16,456,108	100.0	34	8,958	68	1.0	0	123	102.5
		LSTM	5	100.0	920,385	100.0	11	8,320	22	1.0	0	30	230.5

1: Density corresponds to the proportion of key data structures recovered versus total data structures found.
2: Number of invalid tensors that have a valid shape and element count, but contain corrupted pointers or a reference count of 0.
3: Number of management objects traversed to reach key C structs.

4.1 Experimental Setup & Datasets

AiP does not require any modifications to the framework for its model recovery and rehosting. To demonstrate AiP's robustness in recovery, we evaluated its capabilities on three different recent versions of PT (1.6, 1.10, 1.11) and TF (2.2, 2.3, 2.4). AiP's rehosting capabilities were evaluated on a machine running Windows 10.0.1 with 16GB memory using an NVIDIA GTX 1070 GPU (i.e., the investigator's system).

Datasets and Models. Our evaluation was done with the LISA [47], CIFAR10 [48], and IMDB [49] datasets. LISA is an annotated traffic sign dataset. We used Resnet152 [57] and MobileNetV1 [58] as our models, implemented in both TF and PT. The CIFAR10 dataset contains 32x32 images for 10 different classes. For CIFAR10, we used VGG16 [59] and MobileNetV2 [60], implemented in both TF and PT. Finally, for the IMDB dataset, containing reviews of movies labeled positive or negative, we used an LSTM-based RNN [61]. For all three datasets, we split the training and testing dataset such that 20% of the dataset was used for testing (10K images for CIFAR10, ∼1.6K images for LISA, 10k reviews for IMDB) and 80% was used for training. Each model was deployed on a system with Debian 11 with 16 GB RAM, NVIDIA GTX 1080 Ti GPU with CUDA 11.2 (driver version 460.91.03).

Memory Image Acquisition. Memory images were collected for all models across all framework versions. Memory snapshots were taken 10–120 seconds following the evaluation of the model and included the entire Debian system's physical memory. GPU memory images were taken immediately after the snapshot of the system's physical memory. We ensure that address space layout randomization (ASLR) and kernel address space layout randomization (KASLR) are enabled and that we use production binaries (stripped and no debug symbols). Information about how memory dumps were collected can be found in Appendix A.

4.2 Model Recovery

We first evaluated AiP's capability of model recovery (Table 2). Columns 1–3 show the framework (TF/PT), dataset, and model used in each evaluation, respectively. Columns 4–7 show the number of layers and weights recovered by AiP for each model type. We observe that each is recovered with 100% accuracy (30/30 models).

Column 8 presents the number of GPU pointers walked in AiP's model recovery (when GPU data is accessed from pointers). The GPU pointer count often exceeds the layer count in each model. Resnet152 has 940 GPU pointers and

522 recovered layers, as each layer contains a set of weights for the bias and kernel of that layer. VGG16 on PT has fewer GPU pointers (34) than layers (38), as operations such as pooling are counted by AiP as a "layer" but do not point to any data.

The number of Python (Column 9) and C (Column 10) objects recovered per run remains similar between different models within the same framework, with a variance of up to around 50K Python objects for the same model but a different framework version. For example, Resnet152 TF models have Python objects recovered ranging from 342,507 to 390,855 and C objects ranging from 8,720 to 9,976. Little variance exists between the number of C objects recovered for the same model on different framework versions (version specific data-structure changes) with a maximum variance of 1,256 C objects for Resnet152 on TF 2.4 and 2.3.

Though each framework version is different from each other, we found that adding additional versions required 3-5 extra data structure definitions, making the total change in C objects recovered to be small. We observe that in recovered PT models, there is no variance between the number of recovered C objects (i.e., VGG16 has 68 and MobileNetV2 has 640 C objects for all versions). This indicates that once the PT model root is found, the process of recovering key data structures is straightforward for AiP (direct pointer traversal).

TensorFlow vs. PyTorch. While evaluating AiP on the TF and PT frontends, we observed trends for each framework (Columns 11-14). We define density (Column 11) as the proportion of key data structures (structures needed for rehosting) recovered versus total data structures found. A lower density does *not* indicate poor performance by AiP; in fact, it indicates that AiP was able to filter out invalid data structures and recover the necessary data structures for rehosting. All data structures recovered from PT are valid (density of 1.0 across all evaluations), while AiP had to filter out invalid tensors and invalid handles during the recovery for TF (avg density of 0.56). This difference in density can be attributed to the different implementations of the underlying frameworks for PT and TF. Particularly, the major difference between frameworks and associated memory forensics frontends is the necessary management object implementations leading to tensor root objects. TF's tensor objects are less tightly linked by pointers to the model object than PT's tensor objects are, leading to invalid tensors being found by AiP (and discarded) for TF.

Our reasoning is demonstrated in Column 12. As a result of AiP's deterministic nature in finding all model data structures in PT, AiP filters no invalid tensors during its recovery. In contrast, AiP encounters a high number of invalid tensors for TF. The most invalid tensors were seen in the experiment with Resnet152 on TF 2.4 (849), while the fewest were seen with the BD-LSTM on TF 2.2 and 2.3 (8). These results show that even with snapshots where invalid tensor data structures are present, AiP can distinguish valid/invalid nodes, resulting in accurate model recovery.

Management Objects. Column 13 shows the number of management objects in each memory forensics frontend traversed by AiP to find the low-level C tensor objects necessary for recovery. With a greater number of layers, there is an increase in the number of management objects for recovery. For example, Resnet152 in TF 2.2 (522 layers) required 5,375 management objects to be walked, whereas VGG16 in TF 2.2 (25 layers) only required 190 management objects. The difference in management object count between the same models on different framework versions can be accounted for by the data structures being different between versions. Overall, the number of management objects in the TF memory forensics frontend is greater than PT's, making pointer traversal more complex in TF.

Runtime. The runtime for AiP's recovery in Column 14 is influenced by the difference in tensor object management for each framework. We observe that for the same model with the exact same number of weights (VGG16 in both TF and PT with 16M weights), the PT recovery is much faster than the TF recovery. The greatest difference is seen in VGG16, where the PT (ver 1.10) model was recovered in 94.8 seconds and the TF (ver 2.3) model was recovered in 14k seconds, showing a 150 times faster recovery for the PT model. In Column 4, we see that even though the number of layers recovered by AiP for each model in PT is close to the number recovered for TF,AiP's recovery time is still less for PT. This difference is attributed to the way management objects are coupled to tensor objects in each memory forensics frontend.

Scalability to State of the Art Model Sizes. We also observe that the runtime scales well to large models and is not dependent on model size. Resnet152v1 in TF has 94M parameters, which is similar to many SOTA object-detection models such as YoloV7 [62] (37M parameters) or Transformer [63] (110M parameters). Comparing the runtime of AiP for Resnet152 (94.5M parameters) to BD-LSTM (2.7M parameters, 45 times fewer than Resnet152), we see that the runtime of Resnet is around double, meaning that it does not scale linearly with model size difference (not 45 times longer). Even for BD-LSTM in TF 2.3, the runtime (9,755 sec) exceeds that of Resnet (9,550 sec). In TF 2.4, VGG16 had the lowest runtime (1,046 seconds, ten times lower than the next lowest runtime) but has eight times the parameter count and three times the number of layers as BD-LSTM.

4.3 Model Rehosting

For rehosting, we aim to show that the deployed model is the same as the rehosted model. To do so, we evaluate each model on a set of clean test data from its respective dataset. Table 3 presents the detailed results of AiP's rehosting. Column 1 shows the model type rehosted by AiP. Columns 2 (TF) and 6 (PT) shows the framework versions, which correspond to

Table 3: AiP Model Rehosting.

Model	TensorFlow				PyTorch			
	Ver	Accuracy		#$L_t^\dagger$	Ver	Accuracy		#$L_t^\dagger$
		Pre	Post			Pre	Post	
Resnet152v1		97.3%	97.3%	3		97.2%	97.2%	3
MobileNetV1		97.9%	97.9%	4		98.5%	98.5%	4
MobileNetV2	2.2	82.6%	82.6%	4	1.6	64.1%	64.1%	4
VGG16		72.1%	72.1%	2		66.5%	66.5%	2
(BD)-LSTM		84.2%	84.2%	3		79.5%	79.5%	3
Resnet152v1		97.3%	97.3%	3		97.2%	97.2%	3
MobileNetV1		97.9%	97.9%	4		98.5%	98.5%	4
MobileNetV2	2.3	82.6%	82.6%	4	1.10	64.1%	64.1%	4
VGG16		72.1%	72.1%	2		66.5%	66.5%	2
(BD)-LSTM		84.2%	84.2%	3		79.5%	79.5%	3
Resnet152v1		97.3%	97.3%	3		97.2%	97.2%	3
MobileNetV1		97.9%	97.9%	4		98.5%	98.5%	4
MobileNetV2	2.4	82.6%	82.6%	4	1.11	64.1%	64.1%	4
VGG16		72.1%	72.1%	2		66.5%	66.5%	2
(BD)-LSTM		84.2%	84.2%	3		79.5%	79.5%	3

†. #L_t means the number of layer types.

all models recovered in Table 2. Columns 3 (TF) and 7 (PT) show the deployed model accuracy, while Columns 4 (TF) and 8 (PT) show the rehosted model accuracy.

Given the purpose of AiP (accurate recovery and rehosting of deployed model), the deployed model and post-rehosted model must output the same classification result for the same input. If both models (pre and post) are tested with the same dataset, the accuracies of both models should be *exactly equivalent*. For example, for Resnet152v1, the deployed accuracy is 97.3% and the rehosted accuracy is also 97.3% in TensorFlow, meaning that rehosting succeeded for this model. We observe the same across all evaluations. Deployed models and rehosted models show the same accuracies (Columns 3–4 and 7–8), indicating that AiP successfully rehosted the models enabling further forensic investigation.

Columns 5 and 9 (PT) show the number of layer types rehosted by AiP for each model type. For MobileNetV1 and V2 we expect four layer types to be rehosted across all frameworks and versions (DW Conv, PW Conv, Fully Connected, Batch-Normalization), and we see that AiP was able to recover and rehost all four types. For Resnet152 we expect and find that AiP rehosted three different layer types (Conv, Fully Connected, and Batch-Normalization). For VGG16 AiP rehosted two different layer types (Conv, and Fully Connected). Finally, for the BD-LSTM and LSTM, AiP rehosted three different layer types (Embedding, LSTM, Fully Connected), matching what we expect for the architecture of each model. We observe that the number of layer types rehosted by AiP matches the number of layer types for each model. We take these results (equivalent accuracies and layer types) as evidence to show the rehosting capability of AiP.

4.4 AiP Robustness on Online Learning DNNs

We aim to show that AiP is robust in its recovery and rehosting for models that *are constantly updating and inferring* (online learning models). To demonstrate this, we conducted a timing

Table 4: Evaluation of AiP 's Robustness on Model Recovery for Models Actively Being Updated.

MD	TI[1†]	ACC(%)	ASR(%)	Layers	Weights	GPU Ptrs
MobileNetV2	t_0	95.7	0.011	161	6.5M	268
	t_1	95.4	0.006	161	6.5M	268
	t_2	95.6	0.006	161	6.5M	268
	t_3	95.5	0.007	161	6.5M	268
	t_4	95.3	97.9	161	6.5M	268
	t_5	95.5	97.9	161	6.5M	268
	t_6	95.8	97.9	161	6.5M	268
	t_7	95.3	97.9	161	6.5M	268
	t_8	95.1	97.4	161	6.5M	268
	t_9	95.1	97.4	161	6.5M	268
Resnet152	t_0	97.5	0.002	522	94.5M	940
	t_1	97.5	0.000	522	94.5M	940
	t_2	97.6	0.002	522	94.5M	940
	t_3	97.8	0.000	522	94.5M	940
	t_4	97.8	1.000	522	94.5M	940
	t_5	97.9	1.000	522	94.5M	940
	t_6	97.9	1.000	522	94.5M	940
	t_7	98.0	1.000	522	94.5M	940
	t_8	97.7	1.000	522	94.5M	940
	t_9	97.6	1.000	522	94.5M	940

[1] Time interval for when each snapshot was taken. Each interval was 300 seconds apart.

experiment in which AiP recovers and rehosts such models (using the same hardware and setup detailed in §4.1). The models tested (MobileNetV2, Resnet152) were continuously trained on batches of data, during time intervals t_n, where n ranged from 0-9. Each time interval (TI) was separated by 300 seconds. Measurements of the model's accuracy (ACC), the success rate of the attack (ASR), and a memory snapshot were taken at the end of each TI. At t_4, poisoned data with poison rate 10% was introduced (such as in the §2.1) and the model was trained on poisoned data until the end of t_7. For each snapshot taken at each TI, AiP recovers the model from memory and rehosts the model into a live environment such that the ACC and ASR can be compared to the ACC and ASR taken during the model's deployment. Similar to §4.3, the rehosted ACC/ASR matches the deployed model's ACC/ASR every time.

Table 4 shows the results. At each TI (Column 2), we measured the ACC (Column 3) and ASR (Column 4) of the model after AiP's recovery. Importantly, when poisoned data is introduced into each model's learning (t_4-t_7), we see that the ASR climbs to roughly 98% in the MobileNetV2 model and 100% in the Resnet152 model. The ACC is unaffected (remaining at about 95% and 97% for MobileNetV2 and Resnet152, respectively). The ASR for deployed/rehosted models is unchanged when the models resume training on clean data (t_8-t_9), indicating that the learned backdoor persists even when the model is not actively being poisoned.

Columns 5-7 of Table 4 show the layers, weights, and GPU pointers recovered for each TI, respectively. For each TI, these numbers remain the same even though the model was actively learning during all TIs. Similarly, for each model at each TI, the number of layers recovered remained consistent at 161 for MobileNetV2 and 522 for Resnet152v1. Similarly, the weights and number of GPU pointers remained constant for each TI at 6,487,724/94,583,573, and 268/940, respectively. The recovered weights, layers, and GPU Pointers match the numbers seen in Table 2. These results show that AiP is

robust in recovery and reshosting even in an online learning scenario.

5 Discussion

5.1 Extending To New Frameworks

We have implemented two memory forensics frontends for AiP to handle two popular Python-based ML frameworks (for which there were three versions of each), Pytorch [37] (PT) and TensorFlow [36] (TF). However, we acknowledge that though PT and TF are the most popular ML frameworks [64], there are a plethora of other ML frameworks in Python, Java, and a variety of other languages. Utilizing the information introduced in §3.1, as well as in §3.2, it is possible to implement new frontends targeting these frameworks and languages.

As described in §3.1, we conclude that though intermediate objects (management objects used to manage/store higher-level tensor representations) differ from framework to framework, tensor representations all have similar structure and usage. The nodes of each model will remain similar given a new framework, and we saw through TF/PT that the number of different management objects used to get from a high-level representation of a layer to a tensor is under 10. Forensic investigators only need to add new data structure signatures for these management objects to extend AiP to new frameworks. A variety of memory forensics works [65]–[69] also employ similar strategies to extend their methodologies to new frameworks and systems. Fortunately, investigators also have access to multiple data-structure generation tools [41], [70], [71] that can assist with the task of generating structures for a new frontend.

Anti-Forensics Framework Design. Frameworks that stray from the efficient design of low-level tensor objects (obfuscating the generic and required elements to represent a node in a DNN seen in §3.1) can make recovery harder for AiP. However, as discussed in previous works, obfuscated platforms and systems can be analyzed by forensic techniques [39], [41], [44] with extra work by the investigator to use tools to recover the data-structures of obfuscated objects [41], [70], [71]. Investigators can use these tools to recover the data structures for the obfuscated ML framework. Utilizing these data structures, the investigator can then implement the frontend for that framework in the same way as for non-anti-forensics frameworks. Notably, anti-forensics systems would still need to hold the minimum quantity of information necessary to pass to GPU-level functions, as discussed in §2.2, enabling AiP's recovery algorithm.

5.2 Limitations

Secure Enclaves and TEEs. Companies may not want proprietary models trained on sensitive data to be accessible to an adversary intending to pry into their ML system. As a result, defenses towards memory acquisition can be utilized by ML systems. Secure Enclaves (SEs), secure memory regions that can be filled with sensitive training data, model parameters, etc, and Trusted Execution Environments (TEEs) are used to guarantee that training data and model parameters cannot be accessed by adversaries [72]–[74].

Though SEs and TEEs restrict memory acquisition, in the situation where memory from the TEE or secure enclave *is* collected, AiP can still accurately recover the ML model. Restrictions on memory acquisition is an issue that plagues all memory forensics tools/approaches, and is not unique to AiP. AiP, as well as any other memory forensics tools, relies on a complete memory dump to perform model recovery, meaning that given collected memory dumps from the TEE or SE AiP function as intended.

Input Formatting. ML applications have two relevant "inputs" as far as AiP is concerned. First, the application collects raw inputs (e.g., the video feed from a self-driving car). Second, raw inputs are processed before being passed in the proper format (e.g., RGB 640x640x3) to the DNN input layer. To use white-box analyses, the investigator only needs to know the format of the input *passed into the DNN* and not the raw data collected by the application. To help the investigator, AiP will output the buffer associated with the input to the DNN's input layer (e.g., this buffer would contain the pixels of the input image to a DNN). AiP can do this because inputs to the DNN are stored internally as tensors (used during training/inference). AiP's tensor recovery in §3.3 recovers tensors whose shapes match the expected shape of the input layer to the DNN. Upon recovery of these tensors, AiP recovers their tensor buffers, which reveals the input format to the DNN. Inspecting the bytes (e.g., pixels) in the buffer reveals the data's byte-wise ordering, RGB/YUV format, etc. Additionally, it may be helpful for an investigator to recover the application's raw input format. AiP could be extended to use any prior works [75]–[77] to aid in input format analysis once example image buffers are recovered by AiP.

6 Related Work

Memory Forensics. While memory-forensics techniques [78] have been applied to areas such as value-set analysis [79], malware detection/analysis [66], [80], and sequencing user activity [81], no works aim to recover ML models from memory. This is due to the previously discussed complexities regarding interpreted languages like Python, as well as frameworks like Pytorch [37] and TensorFlow [36]. Similarly, while ML models rely on GPUs for model inference and training, no work has investigated object recovery from GPU memory, making ML model recovery an impossibility. Previous work has used CPU memory dumps to infer how malware may utilize the GPU [67], but no work

to our knowledge creates a link between CPU and GPU memory to recover objects from both. Another stem of research studies [66], [82] has deployed ML to aid in memory forensics, which is distinct from our study proposing memory forensics of ML systems.

Attacks Against DNNs. Attacks targeting DNNs are diversifying and becoming more sophisticated. Poisoning attacks use adversary-crafted input during training to cause misclassification of inputs in deployment [83], [84]. Poisoning attacks can have negative effects on the process of Federated Learning and Transfer Learning [14]. With the advent of uniquely deployed models and various forms of model refinement in federated and transfer learning, it is clear that the live ML model on a deployed system should be tested if misbehavior occurs. Moreover, as shown in §2.1, DNN models can be forced to misclassify objects with a *backdoor* hidden in a deployed model [14], [85]–[87]. These attacks can be enabled during runtime, necessitating that the live model be extracted for analysis. AiP helps the investigator recover live DNN models in deployed systems to enable detection of such attacks.

7 Conclusion

We propose AiP, a system for recovering and rehosting DL models from device memory images to enable forensic analysis. AiP is the first system capable of navigating the data structures of the OS, interpreter, and ML framework to accurately recover the parameters that define a model. AiP then rehosts the model within a live process on the investigator's device for further investigation. AiP when evaluated on five unique DL model types across two frameworks, successfully recovered and rehosted all models.

8 Acknowledgements

We thank the anonymous reviewers for their constructive comments and feedback. In particular, we thank our shepherd for their guidance throughout the revision process. We also thank our collaborators at the Georgia Tech Research Institute (GTRI), especially Noah Tobin, for their support, insights, and suggestions throughout this research. This material was supported in part by the GTRI Graduate Research Fellowship Program, as well as the Office of Naval Research (ONR) under grants N00014-19-1-2179 and N00014-23-1-2073. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors and collaborators.

References

[1] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: From phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016. [Online]. Available: https://arxiv.org/abs/1605.07277.

[2] P. McDaniel, N. Papernot, and Z. B. Celik, "Machine learning in adversarial settings," in *Proc. 37th IEEE Security and Privacy*, May 2016.

[3] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proc. European Symposium on Security and Privacy (EuroS&P)*, Mar. 2016.

[4] S. Samanta and S. Mehta, "Towards crafting text adversarial samples," *arXiv preprint arXiv:1707.02812*, 2017. [Online]. Available: https://arxiv.org/abs/1707.02812.

[5] U. Jang, X. Wu, and S. Jha, "Objective metrics and gradient descent algorithms for adversarial examples in machine learning," in *Proc. 33rd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2017.

[6] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, "Detecting adversarial samples from artifacts," *arXiv preprint arXiv:1703.00410*, 2017. [Online]. Available: https://arxiv.org/abs/1703.00410.

[7] N. Das, M. Shanbhogue, S.-T. Chen, F. Hohman, S. Li, L. Chen, M. E. Kounavis, and D. H. Chau, "Shield: Fast, practical defense and vaccination for deep learning using jpeg compression," in *Proc. 24th ACM KDD*, Aug. 2018.

[8] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint arXiv:1704.01155*, 2017. [Online]. Available: https://arxiv.org/abs/1704.01155.

[9] Z. Meng, J. Li, Y. Gong, and B.-H. Juang, "Adversarial teacher-student learning for unsupervised domain adaptation," in *Proc. 2018 International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Apr. 2018.

[10] X. Zhu, "An optimal control view of adversarial machine learning," *arXiv preprint arXiv:1811.04422*, 2018. [Online]. Available: https://arxiv.org/pdf/1811.04422.pdf.

[11] A. Tarvainen and H. Valpola, "Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results," in *Proc. 32nd NeurIPS*, Dec. 2018.

[12] M. Abramson, "Toward adversarial online learning and the science of deceptive machines," in *Proc. 2015 AAAI Conference on Artificial Intelligence (AAAI)*, Nov. 2015.

[13] Z. Shen, Z. He, and X. Xue, "Meal: Multi-model ensemble via adversarial learning," in *Proc. 33rd AAAI Conference on Artificial Intelligence (AAAI)*, Jan. 2019.

[14] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *arXiv preprint arXiv:1708.06733*, 2017. [Online]. Available: https://arxiv.org/abs/1708.06733.

[15] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," in *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2018.

[16] P. Kairouz, H. B. McMahan, B. Avent, *et al.*, "Advances and open problems in federated learning," *arXiv preprint arXiv:1912.04977*, 2019. [Online]. Available: https://arxiv.org/abs/1912.04977.

[17] M. Fang, X. Cao, J. Jia, and N. Gong, "Local model poisoning attacks to byzantine-robust federated learning," in *Proc. 29th USENIX Security*, Aug. 2020.

[18] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *Proc. 29th USENIX Security*, Aug. 2020.

[19] R. Stevens, O. Suciu, A. Ruef, S. Hong, M. Hicks, and T. Dumitraş, "Summoning demons: The pursuit of exploitable bugs in machine learning," *arXiv preprint arXiv:1701.04739*, 2017. [Online]. Available: https://arxiv.org/abs/1701.04739.

[20] Y. Dong, X. Yang, Z. Deng, T. Pang, Z. Xiao, H. Su, and J. Zhu, "Black-box detection of backdoor attacks with limited information and data," *arXiv preprint arXiv:2103.13127*, [Online]. Available: https://arxiv.org/abs/2103.13127.

[21] Y. Liu, W.-C. Lee, G. Tao, S. Ma, Y. Aafer, and X. Zhang, "Abs: Scanning neural networks for back-doors by artificial brain stimulation," in *Proc. 26th ACM CCS*, Nov. 2011.

[22] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," in *Proc. 40th IEEE Security and Privacy*, May 2019.

[23] A. Veldanda Kumar, K. Liu, B. Tan, P. Krishnamurthy, F. Khorrami, R. Karri, B. Dolan-Gavitt, and S. Garg, "Nnoculation: Broad spectrum and targeted treatment of backdoored dnns," *arXiv preprint arXiv:2002.08313*, 2020. [Online]. Available: https://arxiv.org/abs/2002.08313.

[24] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. Molloy, and B. Srivastava, "Detecting backdoor attacks on deep neural networks by activation clustering," *arXiv preprint arXiv:1811.03728*, 2018. [Online]. Available: https://arxiv.org/abs/1811.03728.

[25] K. Liu, B. Dolan-Gavitt, and S. Garg, "Fine-pruning: Defending against backdooring attacks on deep neural networks," in *Proc. 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Sep. 2018.

[26] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *IEEE Transactions on Evolutionary Computation*, 2019. DOI: 10.1109/TEVC.2019.2890858.

[27] *Churning out machine learning models: Handling changes in model predictions*, https://malware.news/t/churning-out-machine-learning-models-handling-changes-in-model-predictions/28755, [Accessed: 2023-01-19].

[28] *Cybersecurity portfolio for business*, https://kaspersky.antivirus.lv/files/media/en/enterprise/Cybersecurity-Portfolio-for-Business.pdf, [Accessed: 2023-01-19].

[29] *When big ai labs refuse to open source their models, the community steps in*, https://techcrunch.com/2022/05/19/when-big-ai-labs-refuse-to-open-source-their-models-the-community-steps-in/, [Accessed: 2023-01-19].

[30] R. Wu, T. Kim, D. Tian, A. Bianchi, and D. Xu, "DnD: A Cross-Architecture deep neural network decompiler," in *Proc. 31st USENIX Security*, Aug. 2022.

[31] Z. Liu, Y. Yuan, S. Wang, X. Xie, and L. Ma, "Decompiling x86 deep neural network executables," in *Proc. 32nd USENIX Security*, Aug. 2023.

[32] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight(s): A large-scale study on insufficient machine learning model protection in mobile apps," in *Proc. 30th USENIX Security*, Aug. 2021.

[33] *About Face ID advanced technology*, https://support.apple.com/en-us/HT208108, [Accessed: 2023-01-19].

[34] *The five pillars of tesla's large-scale fleet learning*, https://medium.com/@strangecosmos/the-five-pillars-of-teslas-large-scale-fleet-learning-approach-to-autonomous-driving-9f6a67aa2d0b, [Accessed: 2023-01-19].

[35] I. Goodfellow and N. Papernot, *The challenge of verification and testing of machine learning*, [Accessed: 2023-01-19]. [Online]. Available: http://www.cleverhans.io/security/privacy/ml/2017/06/14/verification.html.

[36] M. Abadi, A. Agarwal, P. Barham, *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," *arXiv preprint arXiv:1603.04467*, 2016. [Online]. Available: https://arxiv.org/abs/1603.04467.

[37] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. 33rd NeurIPS*, Dec. 2019.

[38] *Cuda, release: 10.2.89*, https://developer.nvidia.com/cuda-toolkit, [Accessed: 2023-01-19].

[39] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proc. 16th ACM CCS*, Nov. 2009.

[40] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proc. 16th ACM CCS*, Nov. 2009.

[41] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proc. 18th NDSS*, Feb. 2011.

[42] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, "Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation*, pp. 197–210, 2006. DOI: 10.1016/j.diin.2006.10.001.

[43] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Vcr: App-agnostic recovery of photographic evidence from android device memory images," in *Proc. 22nd ACM CCS*, Oct. 2015.

[44] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "Dscrete: Automatic rendering of forensic information from memory images via application logic reuse," in *Proc. 23rd USENIX Security*, Aug. 2014.

[45] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Proc. 2010 IEEE International Symposium on Circuits and Systems*, May 2010.

[46] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," *Nature*, pp. 533–536, 1986. DOI: 10.1038/323533a0.

[47] S. Sivaraman and M. M. Trivedi, "A general active-learning framework for on-road vehicle recognition and tracking," *Transactions on Intelligent Transportation Systems*, 2010. DOI: 10.1109/TITS.2010.2040177.

[48] *Cifar-10 (canadian institute for advanced research)*, http://www.cs.toronto.edu/~kriz/cifar.html, [Accessed: 2023-01-19].

[49] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proc. 2011 Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Jun. 2011.

[50] J. Guo, A. Li, and C. Liu, "AEVA: Black-box backdoor detection using adversarial extreme value analysis," in *Proc. 10th ICLR*, Apr. 2022.

[51] Y. Liu, M. Shiqing, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," in *Proc. 2018 NDSS*, Feb. 2018.

[52] J. Stempel, *Jury finds tesla 1% negligent in fatal model s crash*, [Accessed: 2023-01-19]. [Online]. Available: https://www.reuters.com/business/autos-transportation/jury-finds-tesla-just-1-liable-owes-105-mln-over-fatal-crash-2022-07-19/.

[53] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, pp. 135–151, 2002. DOI: 10.1145/567806.567807.

[54] *TensorFlow Core - tf.Tensor*, https://www.tensorflow.org/api_docs/python/tf/Tensor, [Accessed: 2023-01-19].

[55] *PyTorch - torch.Tensor*, https://pytorch.org/docs/stable/tensors.html, [Accessed: 2023-01-19].

[56] O. R. developers, *Onnx runtime*, https://onnxruntime.ai/, [Accessed: 2023-01-19].

[57] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016. DOI: 10.1109/TPAMI.2016.2577031.

[58] *Ssd mobilenet v1 object detection with fpn feature extractor*, https://tfhub.dev/tensorflow/ssd_mobilenet_v1/fpn_640x640, [Accessed: 2023-01-19].

[59] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014. [Online]. Available: https://arxiv.org/abs/1409.1556.

[60] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. 2018 IEEE/CVF CVPR*, Jun. 2018. DOI: 10.1109/CVPR.2018.00474.

[61] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 1997. DOI: 10.1162/neco.1997.9.8.1735.

[62] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," in *Proc. 2023 IEEE/CVF CVPR*, Jun. 2023.

[63] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. 31st NeurIPS*, Dec. 2017.

[64] *Top 10 python packages for machine learning*, https://www.activestate.com/blog/top-10-python-machine-learning-packages/, [Accessed: 2023-01-19].

[65] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Guitar: Piecing together android app guis from memory images," in *Proc. 22nd ACM CCS*, Oct. 2015.

[66] R. Petrik, B. Arik, and J. M. Smith, "Towards architecture and os-independent malware detection via memory forensics," in *Proc. 25th ACM CCS*, Oct. 2018.

[67] D. Balzarotti, R. Pietro, and A. Villani, "The impact of gpu-assisted malware on memory forensics: A case study," in *Proc. 2015 Digital Forensic Research Conference (DFRWS)*, Aug. 2015.

[68] W. Song, H. Yin, C. Liu, and D. Song, "Deepmem: Learning graph neural network models for fast and robust memory forensic analysis," in *Proc. 25th ACM CCS*, Oct. 2018.

[69] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III, "Screen after previous screens: Spatial-temporal recreation of android app displays from memory images.," in *Proc. 25th USENIX Security*, Aug. 2016.

[70] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled Reverse Engineering of Types in Binary Programs," in *Proc. 18th NDSS*, Feb. 2011.

[71] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, "Dimsum: Discovering semantic data of interest from un-mappable memory with confidence," in *Proc. 19th NDSS*, Feb. 2012.

[72] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "Darknetz: Towards model privacy at the edge using trusted execution environments," in *Proc. 18th ACM International Conference on Mobile Computing Systems (MobiSys)*, Jun. 2020.

[73] D. L. Quoc, F. Gregor, S. Arnautov, R. Kunkel, P. Bhatotia, and C. Fetzer, "Securetf: A secure tensorflow framework," in *Proc. 21st ACM/IFIP/USENIX Middleware Conference*, Dec. 2020.

[74] Z. Sun, R. Sun, C. Liu, A. R. Chowdhury, L. Lu, and S. Jha, "Shadownet: A secure and efficient on-device model inference system for convolutional neural networks," in *Proc. 44th IEEE Security and Privacy*, May 2023.

[75] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[76] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proc. 15th ACM CCS*, Oct. 2008.

[77] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proc. 24th ACM CCS*, Oct. 2017.

[78] F. Pagani and D. Balzarotti, "Back to the whiteboard: A principled approach for the assessment and design of memory forensic techniques," in *Proc. 28th USENIX Security*, Aug. 2019.

[79] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis," in *Proc. 28th USENIX Security*, Aug. 2019.

[80] M. Yao, J. Fuller, R. P. Sridhar, S. Agarwal, A. K. Sikder, and B. Saltaformaggio, "Hiding in plain sight: An empirical study of web application abuse in malware," in *Proc. 32nd USENIX Security*, Aug. 2023.

[81] R. Bhatia, B. Saltaformaggio, S. J. Yang, A. I. Ali-Gombe, X. Zhang, D. Xu, and G. G. Richard III, "Tipped off by your memory allocator: Device-wide user activity sequencing from android memory images.," in *Proc. 2018 NDSS*, Feb. 2018.

[82] W. Song, H. Yin, C. Liu, and D. Song, "Deepmem: Learning graph neural network models for fast and robust memory forensic analysis," in *Proc. 25th ACM CCS*, Oct. 2018.

[83] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," Jun. 2012. DOI: 10.5555/3042573.3042761.

[84] L. Muñoz-González, B. Biggio, A. Demontis, A. Paudice, V. Wongrassamee, E. C. Lupu, and F. Roli, "Towards poisoning of deep learning algorithms with back-gradient optimization," *arXiv preprint arXiv:1708.08689*, 2017. [Online]. Available: https://arxiv.org/abs/1710.00942.

[85] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *arXiv:1801.00553*, 2018. [Online]. Available: https://arxiv.org/abs/1801.00553.

[86] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," in *Proc. 25th ACM CCS*, Oct. 2018.

[87] Y. Adi, C. Baum, M. Cisse, B. Pinkas, and J. Keshet, "Turning your weakness into a strength: Watermarking deep neural networks by backdooring," in *Proc. 27th USENIX Security*, Aug. 2018.

[88] Microsoft, *Microsoft/avml: Avml*, [Accessed: 2023-01-19]. [Online]. Available: https://github.com/microsoft/avml.

[89] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why Should I Trust You?: Explaining the Predictions of Any Classifier," *arXiv preprint arXiv:1602.04938*, 2016. [Online]. Available: https://arxiv.org/abs/1602.04938.

[90] Berla, *Berla*, [Accessed: 2023-01-19]. [Online]. Available: https://berla.co/.

[91] T. Holt and D. S. Dolliver, "Exploring digital evidence recognition among front-line law enforcement officers at fatal crash scenes," *Forensic Science International: Digital Investigation*, vol. 37, 2021. DOI: 10.1016/j.fsidi.2021.301167.

[92] K. K. Gomez Buquerin, C. Corbett, and H.-J. Hof, "A generalized approach to automotive forensics," *Forensic Science International: Digital Investigation*, vol. 36, 2021. DOI: 10.1016/j.fsidi.2021.301111.

[93] C. Duboka, "Considerations in forensic examination of automotive systems," *Int. J. of Forensic Engineering*, pp. 111–130, 2012. DOI: 10.1504/IJFE.2012.050408.

[94] N.-A. Le-Khac, D. Jacobs, J. Nijhoff, K. Bertens, and K.-K. R. Choo, "Smart vehicle forensics: Challenges and case study," *Future Generation Computer Systems*, vol. 109, 2020. DOI: 10.1016/j.future.2018.05.081.

[95] M. H. Rais, R. A. Awad, J. Lopez, and I. Ahmed, "Jtag-based plc memory acquisition framework for industrial control systems," *Forensic Science International: Digital Investigation*, vol. 37, 2021. DOI: 10.1016/j.fsidi.2021.301196.

[96] N. Zubair, A. Ayub, H. Yoo, and I. Ahmed, "Pem: Remote forensic acquisition of plc memory in industrial control systems," in *Proc. 2022 Digital Forensic Research Conference (DFRWS)*, Mar. 2022.

[97] H. Yoo, S. Kalle, J. Smith, and I. Ahmed, "Overshadow plc to detect remote control-logic injection attacks," in *Proc. 2019 Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Jun. 2019.

[98] *Cuda-gdb*, https://docs.nvidia.com/cuda/cuda-gdb/index.html#cuda-gdb-extensions, [Accessed: 2023-01-19].

A Memory Image Acquisition

A variety of tools and techniques assist forensic investigators in acquiring memory images. Microsoft's AVML [88] and LiME [89] are popular tools for recovering volatile memory from Linux. Memory acquisition from Android devices is possible through LiME (enabling the analysis of models deployed on a smartphone via AiP's recovery). Forensic techniques have been developed to analyze automotive systems/automotive system memory [90]–[94]. Likewise, embedded systems such as Arduino and Raspberry Pi come with capabilities to dump their own volatile memory. Even PLCs, considered critical infrastructure in ICS, have had tools designed to acquire and analyze their volatile memory [95]–[97] for security analysis.

For the experiments conducted in this paper, the memory images (CPU and GPU) given to AiP are collected from a running ML system. To collect a CPU memory image, we utilize LiME [89]. LiME produces a complete image of volatile memory from the system. This image contains all memory from running processes, including the process with the live DNN. Once the CPU memory image is collected, we collect the GPU memory of the running Python process. This memory image, primarily used for debugging, contains information pertaining to GPU registers, device/grid/block/warp/thread info, and the global memory associated with the process. To collect the GPU memory image, CUDA-GDB's [98] manual GPU core image generation feature is used. At the start of process execution, when communication with the GPU occurs, a pipe is created by CUDA. Writing to this pipe during process execution stops the process and produces a GPU memory image containing the GPU memory at the moment when the pipe was written to. By utilizing both memory images, AiP is able to perform its model recovery and model rehosting.