Achieving Zen: Combining Mathematical and Programmatic Deep Learning Model Representations for Attribution and Reuse

David Oygenblik*, Dinko Dermendzhiev*, Filippos Sofias*, Mingxuan Yao*, Haichuan Xu*, Runze Zhang*, Jeman Park†, Amit Kumar Sikder‡ and Brendan Saltaformaggio*
* Georgia Institute of Technology
† Kyung Hee University
‡ Iowa State University

***Abstract*—Prior work has developed techniques capable of extracting deep earning (DL) models in universal formats from system memory or program binaries for security analysis. Unfortunately, such techniques ignore the recovery of the DL model's programmatic representation required for model reuse and any white-box analysis techniques. Addressing this, we propose a novel recovery methodology, and prototype ZEN, that automatically recovers the DL model programmatic representation complementing the recovery of the mathematical representation by prior work. ZEN identifies novel code in an unknown DL system relative to a base model and generates patches such that the recovered DL model can be reused. We evaluated ZEN on 21 SOTA DL models, including models across the language and vision domains, such as Llama 3 and YoloV10. ZEN successfully attributed custom models to their base models with 100% accuracy, enabling model reuse.**

I. Introduction

Proprietary deep learning (DL) models are often extended from open-source state-of-the-art models [1]–[4]. At the same time, the research community has proposed numerous model analysis tools for backdoor detection [5]–[8], model explainability [9], intellectual property protection [10], etc. However, all of these techniques strongly assume *white-box model access* [11]–[13]. This assumption often *only holds for the original developer of the proprietary model*, as only they have white-box access (i.e., knowledge of the model's architecture, weights, custom layer implementations, etc.). This white-box assumption will rarely hold for forensic investigators who aim to apply model analysis tools to proprietary models post-deployment [11], [13]. Without cooperation from the original developer, these models are *black-box* (i.e., unknown architecture, dependencies, and custom layer implementations). To apply any of those white-box model analysis techniques, forensic investigators need a method to bridge the black-box to white-box gap.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231628
www.ndss-symposium.org

Existing work aims to convert black-box DL models to white-box through methodologies such as operator loop analysis [14], learning-based approaches [15], symbolic analysis [14], [15], or memory forensics [16]. Such approaches successfully recover an operator's mathematical representation, such as tensors, weights, and biases, and identify basic layer types based on this representation. However, we find that these techniques directly ignore the DL model's **code**, making model-recovery, environment setup, and model re-execution/instrumentation impossible for models containing *any* customized code (as shown in Table V). Even approaches [14] that attempt to export recovered artifacts in the universal ONNX format [17] would fail to enable model re-execution, as ONNX, while prebuilt with standard operators, strictly requires a set of functions for customized code or novel operator implementations to enable model execution. Ultimately, this means that the application of model-testing techniques that rely on instrumentation of the model's code are directly inapplicable. Prior research [5], [7] and our preliminary study (§II-A) have shown that without access to the instrumentable code of the deployed DL model, prior approaches will fail to satisfy the assumptions necessary to enable white-box analysis.

Applying white-box analysis tools [5], [7], [8], [18]–[20] to proprietary DL models faces several challenges. First, enabling model reuse for code instrumentation is essential but hindered by the difficulty of recreating the correct execution environment, as prior work directly ignores this. Second, environment setup relies on an investigator's knowledge of what model is being deployed, what dependencies it requires, and what framework it can be deployed in. To answer these questions, an investigator can attempt to **attribute** the model to identify the original base model, yet customization often obscures a model's lineage, making attribution difficult. Relying solely on layer names, weights, and model graph structure for comparison is insufficient, as significant variations exist even within model families (shown in §IV-B1). Third, robust attribution and reuse require recovering a "fingerprint" of the model, which is the code that implements the DL model. Without this fingerprint, the necessary context

for handling custom implementations and enabling reuse for white-box analysis is lost. These issues highlight that *a DL model cannot be analyzed in isolation from its implementing code*, necessitating an approach that synthesizes both its mathematical and programmatic representations.

We find that bridging the black-box to white-box gap for proprietary DL model analysis hinges on three key insights that address the challenges of code recovery and model attribution/reuse. First, by recovering not only the model's layers/weights/graph structure but also its implementing code, it is possible to create a unified representation (or fingerprint) of the model. This fingerprint captures both the *what* and the *how* of the model's operation, supplying the code context missing from prior approaches. Second, this model fingerprint, enables robust model attribution. Leveraging the common practice of building upon open-source base models [1]–[4], the fingerprint of the unknown model can be compared against a library of known base model fingerprints to identify the proprietary model's origin. Third, model **reuse** can be achieved through differential analysis and patching. By comparing fingerprints between the recovered model and the attributed base model, specific customizations (code additions or modifications) can be identified. These differences can then be translated into code patches that are applied via runtime techniques to an instance of the base model to recreate the recovered model's functionality in an environment suitable for white-box analysis.

Drawing upon these insights, we propose ZEN, an automated model-recovery framework that enables the application of white-box analysis tools to black-box models post-deployment. ZEN is the first work to synthesize a DL model's unified representation, mapping DL operators to user-defined code that runs in the DL system (§IV-A). ZEN then aims to attribute the deployed model to a single base model from which code differences can be applied for model reuse (§IV-B1). ZEN comes prebuilt with a *base model library*, which contains fingerprints of vision/language models from which model attribution can be performed. With a base model identified, ZEN finds differences in code between the two models, preparing patches for model redeployment. Finally, given the base model environment, ZEN instruments the live-runtime model environment with generated patches, enabling model reuse *without* source code access to the deployed DL system.

We evaluate ZEN on 21 different model types spanning the language/vision model domains (e.g, Llama 3 [21], YoloV10 [22]), to show that ZEN was able to accurately recover the unified representations for all 21 models. ZEN was then able to correctly attribute each to a base model even when the model had been customized by up to 83.3%. Using the attributed model, ZEN is able to identify all customized changes, enabling patching, model reuse, and white-box analysis. To facilitate future work and recovery techniques we will directly open source our code as well as "model fingerprint" hosting web service (§II-B).[1].

II. A Unified Model Representation

Prior work [14]–[16], [23], [24] struggles to enable realistic model reuse. First, they assume a working reuse environment for recovered models. Proprietary models can rely on esoteric third-party libraries, customized operators, and system-specific pre-/post-processing logic, which prior work is unable to recover. By attributing the DL system to a base model, ZEN can recreate a model environment (§IV-B) where the recovered model can be used. Second, prior work attempts to export the model mathematical representation in ONNX for reuse. As ONNX requires a set of defined custom operators when dealing with unknown layer types, such approaches fail to enable model reuse, as they do not recover the ONNX-required customized operators. To address this, ZEN recovers the programmatic representation of the recovered DL model, providing necessary implementations for model reuse (§IV-C). Finally, as a result of being able to recover only common or inferable operators, those prior techniques are unable to provide instrumentable models for models employing customized operators, impeding the application of vetting techniques. MXR and MXT [23] assume recovered DL model instrumentability, although they ignore custom implementation recovery. ZEN applies discovered customized changes to an attributed base model (§IV-D) to provide an instrumentable model.

A. Motivating Investigation

Consider a mission-critical DL system (e.g. a UAV [25]), built upon state-of-the-art models [1]–[4], [26]. The UAV's DL model utilizes customized model layers and has uniquely trained parameters for its deployment environment. Prior to the widespread deployment of the DL system, an investigator is called in to verify that the UAV is not vulnerable to traditional backdoor/adversarial attacks [27], [28]. Unfortunately, the UAV's model binary has been protected such that it is difficult to reverse engineer (weights encrypted, architecture varied, generic implementation altered). To overcome this, the investigator elects to use model-recovery tools [14]–[16], [23], to analyze the UAV's model. To this end, the investigator uses AIP [16], successfully recovering 230 tensors, 34 layers, and 9.3M weights corresponding to the model's mathematical representation.

a) Model Reuse: The investigator then attempts to lift the recovered model data into a universal reusable format (i.e., ONNX representation). The investigator finds that this is a straw-man solution, as to even generate this representation ONNX requires a set of definitions for each layer in the model. However, definitions for recovered layers (layer code) are not output by the model-recovery tool, and thus fail to satisfy this preliminary assumption. Unfortunately, without addressing this error (supplementing the testing environment with missing code), model testing would be impossible.

[1] *Open sourced upon paper acceptance.*

TABLE I: ZEN's Automated Unified Model Representation Recovery, Base Model Identification, and Patch Generation/Application for Reuse of the UAV Model.

ZEN Input	DL System Memory Image
Mathematical Representation Recovery	**Tensors:** 230 **Layers:** 34 **Weights:** 9.3M
Runtime Recovery	**DL System Code:** **Functions:** 59,054 **Classes:** 1,109 **Modules:** 2,983
Model Code Recovery	**DL Model Code:** **Functions:** 342 **Classes:** 22 **Modules:** 22
Model Attribution	**Similarity Score (SS) Analysis:** Mathematical Representation SS: 0.82 Programmatic Representation SS: 0.98 Unified Representation SS: 0.94 **Base Model Identified:** `YoloV5` [29]
Patch Generation	**Classes Added:** 9 **Changed Functions:** 3 **Added Functions:** 8
Model Reuse	**Mean Average Precision@0.5:** 0.413 **Attack Success Rate:** 82.3%

b) Model Attribution: To determine what needs to be supplemented to the testing environment for model reuse, given AIP output, the investigator can attempt to make an educated *guess* as to the attribution of the model (as attribution is ignored by prior work). By attempting to attribute the model, the investigator can then try to manually identify the implementation changes that need to be added to the testing environment to address runtime errors. Unfortunately, many models share similar architectural backbones and layer types. Even models of the same family (i.e., YoloV5 [29]) can have different weight counts, layer sizes, and additional layers added, depending on the deployment scenario, making guessing a model's attribution difficult.

c) DL System Reverse Engineering: Assuming that the investigator correctly attributed the model to a base model [26], they can then attempt to reverse engineer the UAV to identify the runtime-required code missing in the testing environment. However, this introduces a new set of challenges. First, such DL systems may utilize complex third-party library operators for model inference (i.e., Fairscale [30], LoRA [31], etc.), meaning that without also identifying used third-party library code (from memory), model reuse will remain impossible. Second, upon putting the memory image in a disassembler (i.e., IDA, Ghidra [32], [33]) the investigator sees that there are 59,054 functions and 1,109 classes that manually need to be reverse engineered. Finally, model reuse for testing requires a source-code level instrumentable model. The investigator can attempt to use a Python decompiler [34] to generate that source code but would realize that this approach would produce error-riddled source code [35], [36]. Ultimately, the challenges impeding model reuse are difficult to overcome.

B. Investigating With ZEN

ZEN can be used by the investigator to aid with their UAV testing. Table I highlights the results of ZEN's automated recovery and reuse of the UAV's DL model. ZEN first recovered the model's MR comprised of 9.3 million weights, 34 layers, and 230 tensors, shown in row 2. Likewise, ZEN was able to recover the correct model topology (ordering of layers and layer connections), matching prior work [16].

a) ZEN's DL System Reverse Engineering: Then, ZEN automatically recovers the code of the entire DL system, recovering 59,054 functions, 1,109 classes, and 2,983 modules, shown in row 3 of Table I. Then, by finding the intersection of all code corresponding to the recovered mathematical representation (i.e., the model's layer types), ZEN identifies the model-specific programmatic representation. Shown in row 4 of Table I, ZEN recovers 342 functions, 22 classes, and 22 modules specific to the model. ZEN then synthesizes the recovered mathematical and programmatic representations into a unified representation that acts as a fingerprint for attribution.

b) ZEN's Model Attribution: With the model's unified representation in hand, ZEN employs a novel base model attribution algorithm (§IV-B), calculating the similarity between the unified representation of the recovered model to those of state-of-the-art base models [26]. The investigator finds that the model was identified as a derivative of `YoloV5`, with a similarity score of 0.94 (row 5). Note that by identifying a base model, ZEN can uncover differences between the recovered model and the base model (`YoloV5`), which are subsequently used to generate patches for model reuse. To do this, ZEN utilizes its model differential analysis algorithm (§IV-C), discovering nine unique classes added, three modified functions, and eight added functions to the recovered model (relative to the base model).

c) ZEN's Model Reuse: ZEN then outputs the model's weights/graph structure and a patch file containing all code linked to the layers of the model necessary for downstream model reuse. Given the output of ZEN (base model and patch file), the investigator can download an open-source implementation of the identified model (YoloV5) from an AI marketplace [1]–[3]. Upon setup of the base model's environment, ZEN is then utilized to automatically patch the base model environment to match that of the recovered model, enabling source-code level testing.

Testable model in hand, the investigator evaluates the UAV's model performance on non-perturbed data, ascribing a mAP@0.5 score of 0.413 (row 7 of Table I). The investigator observes that the recovered model outperforms prior YoloV5 models as a result of the model having SOTA layers. Importantly, ZEN enables source-code level instrumentation of the model, allowing for testing with white-box analysis techniques [5], [7], [8], [19]. We applied FGSN [37] as a case study, to highlight the application of white-box analysis techniques. FGSN generated adversarial examples with an 82.3% attack success rate against the UAV's DL model. This shows that post-ZEN's application, even source-code-level instrumentation techniques can be applied to analyze previously black-box DL systems.

III. Threat Model and Assumptions

ZEN was prototyped as a memory forensics tool, and our assumptions regarding its implementation and evaluation mirror the assumptions made in prior memory forensics work [16], [38]–[47]. ZEN operates on a snapshot of physical memory acquired from a running system (acquisition is further discussed in §C) without corruption or tampering of the memory image. In the case of the drone scenario in §II-A, an investigator was able to recover the drone and use a memory-acquisition tool [48]–[50] to acquire a memory image. As ZEN recovers both the code and model weights/layers/graph-structure *at runtime*, any encryption employed to evade static analysis is bypassed. Once the model and model code are loaded, they exist decrypted in memory. However, we do assume (as does prior work) that no hardware-assisted runtime integrity checks or other specialized defenses (e.g., hardware-assisted memory encryption) interfere with the memory-acquisition process.

a) Threat Model: We design ZEN as a tool intended for forensic investigators to analyze unknown, adversarial, or even copyright-infringing DL systems (such as in §V-D). An adversary may attempt to obfuscate their DL system prior to deployment to thwart ZEN (discussed in §B). We assume the adversary deploys Python-based DL systems and discuss extension to C/C++ systems in §VI-A. We exclude other ML models (e.g., regressions, random forests, etc.) from this work, though ZEN's unified model representation supports the recovery/reuse of such models, as they consist of user-defined code and implementation-specific weights/graph structure.

DL model obfuscations, such as layer renaming, dummy code declaration, and control flow obfuscation, are possible but are insufficient to thwart ZEN. This is because obfuscated DL systems are still bound to invoke standard operators required by underlying APIs (e.g., CUDA [51], cuBLAS [52], etc.) corresponding to the layers in the DL model (e.g., convolutions, activations, etc.). ZEN targets the recovery of the code corresponding to these layers in the model's directed graph [16], which contains the lower-level framework calls necessary for the DL model's operation.

We assume the target DL system utilizes common DL frameworks (e.g., PyTorch [53], CUDA [51], etc.) rather than an entirely new framework developed by an adversary. If an adversary designs a new DL framework from scratch, the data structures upon which ZEN relies would be absent and impede recovery. We consider this scenario out of the scope of this work but discuss possible solutions in §VI-C.

We also assume that the adversary's DL framework executes on a known OS with known GPU hardware. Similarly, we consider building new OSes/GPU drivers to be out of the scope of this work, as this would entail extending the underlying tooling that ZEN relies on (e.g., AIP [16]).

IV. ZEN Design

First, we formally define a DL system's unified representation such that generic model recovery and reuse are possible. Let $\mathcal{M}$ represent a DL model. $\mathcal{M}$ is defined as:

$$\mathcal{M} := (\text{MR}, \text{PR}, \phi, \mathcal{E}) \tag{1}$$

where MR and PR are the model's Mathematical Representation and Programmatic Representation, respectively, ϕ is a mapping function between elements in MR and PR, and $\mathcal{E}$ is the execution context. MR is further defined as:

$$\text{MR} := (\mathbf{T}, \mathbf{W}, \mathbf{B}, \mathcal{L}, \mathcal{G}) \tag{2}$$

where $\mathbf{T}$ denotes the set of tensors, $\mathbf{W}$ the set of weights, $\mathbf{B}$ the set of biases, $\mathcal{L}$ the set of layer types, and $\mathcal{G}$ the directed acyclic graph (DAG) defining the connections between layers in the model. Similarly, the PR is defined as:

$$\text{PR} := (\mathcal{C}, \mathcal{F}, \mathcal{R}) \tag{3}$$

where $\mathcal{C}$, $\mathcal{F}$, $\mathcal{R}$ are the set of classes, functions, and data structures, respectively, implementing the layers, connections, and other operations used by the model. The mapping function $\phi : \text{MR} \rightarrow \text{PR}$ establishes the correspondence between elements in MR and PR. For each tensor $t \in \mathbf{T}$ belonging to a specific layer $l \in \mathcal{L}$, $\phi(t) = y$, where $y \in \mathcal{C} \cup \mathcal{F}$. The function ϕ satisfies the following constraints:

$$\forall y_1, y_2 \in \text{PR},\ y_1 \neq y_2 \implies \phi^{-1}(y_1) \cap \phi^{-1}(y_2) = \emptyset. \tag{4}$$

This ensures that no two distinct elements in PR map to the same element in MR, thereby adhering to the DAG $\mathcal{G}$ representing the model. Each node in $\mathcal{G}$ (from MR) maps to exactly one element in the PR. Additionally, ϕ allows multiple tensors $t_1, t_2, \ldots \in \mathbf{T}$ from the same layer in MR to map to a single programmatic element $y \in \text{PR}$, capturing cases such as complex layers requiring multiple tensors for their implementation (e.g., Involution [54], RepConv [55]). Finally, a unified representation can be formulated as:

$$\text{Unified Representation} := (\text{MR}, \text{PR}, \phi). \tag{5}$$

Note that the unified representation does not directly incorporate the execution context $\mathcal{E}$, the runtime environment necessary for the operation of the model. Although $\mathcal{E}$ is critical for executing the model, it is not inherently part of the model's definition but instead defines the model's operational environment. We define the execution context $\mathcal{E}$ as:

$$\mathcal{E} := (\mathcal{H}, \mathcal{D}, \mathcal{S}), \tag{6}$$

where $\mathcal{H}$, $\mathcal{D}$, and $\mathcal{S}$ represent the hardware context (e.g., GPUs, TPUs), dependencies (e.g., libraries/frameworks), and runtime state (e.g., dynamically loaded modules), respectively.

Finally, we define Ψ, which allows for reuse of a model based on its unified representation and $\mathcal{E}$:

$$\Psi : \phi(\text{MR}) \times \mathcal{E} \rightarrow \text{Executable Model}, \tag{7}$$

which is a deployment mapping that combines $\phi(\text{MR})$ and $\mathcal{E}$ to form a deployable artifact (necessary for instrumentation of the model). Note that the executable model produced by Ψ is distinctly different from model $\mathcal{M}$, as $\mathcal{M}$ is the model in its original deployment environment, whereas Ψ aims to enable redeployment and execution of a model's representation in tandem with $\mathcal{E}$ (estimated by attribution in §IV-B).

A. Unified Model Representation Synthesis

Satisfying both Equation 5 and Equation 4, ZEN's unified model representation synthesis is shown in Figure 1. Given a main memory image, ZEN first recovers the mathematical and programmatic representations of the DL system. ZEN adopts existing techniques (① in Figure 1) to recover the model's parameters, layer types, connections, etc. [14]–[16] (②), corresponding to its mathematical representation (Equation 2).

However, recovery of the model's programmatic representation is impeded by a number of technical challenges. First, DL systems run on complex interpreted languages (e.g., Python). DL system source code is compiled at runtime to Python byte-code, or statically to machine code, and is then executed, making source-code level recovery from memory impossible. Second, the recovery of raw byte-code for functions in the DL system is not enough for programmatic representation recovery, as the byte-codes reveal neither the number or types of arguments/local variables nor the classes/modules that functions belong to. Finally, DL systems utilize heavyweight third-party and system-level libraries, making user-defined system implementations hard to distinguish from library function implementations.

We make the key observation that the programmatic representation of the DL model can be created by identifying and recovering code in memory within the closure of the code implementing the DL model. As all DL model node definitions and functions are in memory at runtime, by investigating the current DL system's *code objects* [56] loaded into memory during model execution, ZEN can recover all model-specific code.

a) Model-Specific Code Recovery: To address the first and second challenges, we observe that while source-level implementations are unavailable at runtime, the code objects corresponding to these implementations are available in process memory (as raw bytes). ZEN first identifies *all* code objects in the DL system process memory (③ in Figure 1). ZEN begins by enumerating all code objects within the DL process memory. For each code object, ZEN collects the function signature, class, and address, which are combined by ZEN into an identifier, *function ID* (④) (stored and reused in §IV-D). ZEN then identifies the closure of the associated code object (from function prologue to return), also recursively recovering other functions called by that code object (⑤). ZEN recursively traverses the pointers of the arguments and local variables (⑥) that contain functions and data structures of various types to verify that all functions and data structures used by the code object are not smeared in memory [57], have a type string or Vtable pointer, and have a valid return address in the memory address space (⑦). With this data collected (classes $\mathcal{C}$, functions $\mathcal{F}$, and data structures $\mathcal{R}$), ZEN satisfies Equation 3.

Then, ZEN employs Code Object Filtering (⑧) utilizing recovered model layer types and data structures (highlighted in red), to reduce the set of code objects produced from Runtime Recovery (③) to a set of code objects that solely implement the DL model. ZEN first eliminates any code objects from its

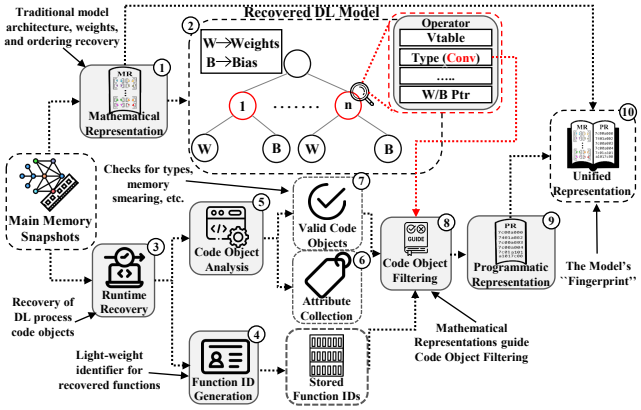

Fig. 1: Given a memory image containing a DL system, ZEN creates a unified model representation, or fingerprint, for downstream attribution and patching.

Runtime Recovery that do not reference objects (or functions) of classes used in the layers of the recovered in ②. ZEN then adds code objects that reference functions or objects of classes in recovered layers to a set constituting the programmatic representation (⑨). However, code objects **not** in this set may still be relevant to the execution context $\mathcal{E}$, as third-party libraries may be used in the deployment of the model (discussed in §IV-B).

b) A Unified Representation: ZEN then synthesizes a unified model representation, combining the mathematical and programmatic representations of the DL model (establishing a mapping function ϕ used in Equation 7). To do this, each node recovered by ZEN for the DL model (e.g., the layers constituting the model's graph $\mathcal{G}$) needs to be mapped to a recovered DL model code object and must satisfy the injectivity constraint in Equation 4. Referring back to the §II-A, the unknown `Involution` node in the DL model has its weights, layer shape, and operator type mapped to the recovered `Involution` class and that class's code objects (i.e., `init` and `forward` functions).

For each node of the DL model's graph $g_i \in \mathcal{G}$, ZEN maps the node based on layer type (e.g., `Involution`) to the associated classes and code objects in the model's programmatic representation. Then, any functions called within the closure of those code objects are added to the mapping for that model node. ZEN repeats this over all layers (nodes) in the DL model's graph. This set of mappings (ϕ) constitutes the unified model representation (⑩), or fingerprint, used in model attribution.

B. Model Attribution

While the unified model representation provides a mapping between the model layers and the code implementing model layers, it is insufficient for model reuse without the execution context $\mathcal{E}$ (e.g., YoloV10 [22] relies on the Ultralytics [58] library for pre-processing, inference, and prediction post-processing). Furthermore, $\mathcal{E}$ also encompasses hardware dependencies and runtime states that are critical for model

deployment. Without a practical mechanism to identify or approximate $\mathcal{E}$, model reuse is impossible.

Worse, recovering $\mathcal{E}$ from memory is inherently challenging due to incomplete information, such as missing external dependencies, transient runtime configurations, and proprietary or obfuscated libraries. Additionally, dependency mismatches, version conflicts, and hardware incompatibilities further complicate rehosting the model's execution context *exactly*. These challenges make precise recovery of $\mathcal{E}$ impractical.

However, our key insight is that the development of proprietary models is often predicated on the adaptation of state-of-the-art, open-source models. By identifying the base model that the proprietary model is built on, ZEN can approximate the execution context $\mathcal{E}$ by the base model's execution context $\hat{\mathcal{E}}$. Referring back to §II-A, by identifying the YoloV5 base model, ZEN can reuse that original YoloV5 execution environment for downstream patching (§IV-D). Gathering an $\hat{\mathcal{E}}$ effectively mitigates dependency issues and provides a pseudo-execution context that supports deployment and reuse of the recovered model.

Basing model attribution solely on the mathematical representation is often infeasible. Due to use-case-specific implementations, models within the same family can exhibit significant mathematical representation divergence (i.e., YoloV8 [59] vs. YoloV8-SPD [60] have a mathematical representation similarity of 0.46 shown in Table III). This is because components of the model (layer sizes, scaling, element counts, and even novel layers) are design-choice specific, meaning that models (such as YoloV8-SPD vs. YoloV8) may have wildly varied element/layer counts, layer types, and layer sizes. Searching for a solution, we looked at attributing a DL system via its unified model representation, or model fingerprint. However, without building a database of fingerprints, there would be no way for an investigator to even attribute an unknown model.

a) Built-in Base Model Library: As foundational models [26] are often modified, improved, fine-tuned, and redeployed for specific use cases, we aim to build a library of such foundational model signatures, which we refer to as the **Base Model Library** (BML), to identify foundational model derivatives. Given a set of foundational models, ZEN synthesizes a unified model representation for each, serving as the fingerprint for that model. To construct the base model library, an investigator can deploy a foundational model, and use a memory-acquisition tool to get a memory image of the DL model system. Given the memory image and the ground truth name of the foundational model as inputs, ZEN applies its methodology (§IV-A) to generate the foundational model's unified representation, storing it as an accessible entry in the base model library (via the ground truth name). By iteratively applying this process to a set of memory images from various DL systems, ZEN constructs a base model library (adding new base models is further discussed in §VI-B).

1) Model Attribution: Even with a base model library, model attribution via unified representation comparison introduces problems that ZEN must solve. Strings, variable names, and filenames (attributes of the DL system and code objects) recovered from the DL system can vary from implementation to implementation. Such attributes are unreliable for comparison, as they are use-case specific and obfuscatable.

Algorithm 1: ZEN's Model Attribution.

Input: Recovered Model Unified Representation (UR) M, BML
Output: Matched Model, UR: MM, Set: Ω, Unmapped COs: Δ

1 $URO \longleftarrow \emptyset$ // Base UR Set Initialization
2 $\Delta \longleftarrow M.co$ // Code Objects from M added to Δ
3 **for** $UR \in BML.urs$ **do**
4 **if** $CO_i.bc \in UR \equiv CO_j.bc \in \Delta$ **then**
 // UR with Match Added to URO.
5 $URO \longleftarrow URO \cup UR$;
6 **end**
7 **end**
8 $MM \longleftarrow \emptyset$ // Initialize UR scores set.
9 **for** $UR_i \in URO$ **do**
10 $MRSS_i \longleftarrow MRSS(UR_i, M)$ // Calculate mathematical representation SS.
11 $\Gamma \longleftarrow UR_i.co$ // Γ Assigned all COs from Candidate UR.
12 $\Gamma.src \longleftarrow Decompile(\Gamma.co.bc)$
13 $\Omega \longleftarrow \emptyset$
14 **do**
15 $\delta_i \longleftarrow \delta \in \Delta$ // Select Code Object from Δ
16 $\delta_i.src \longleftarrow Decompile(\delta_i.bc)$
 // Top COs in Γ with Highest Attr. SS to δ.
17 $TOP_N \longleftarrow argmax_N(Attr(\delta_i.attr, \Gamma))$
 // Byte-code similarity for code objects in TOP_N.
18 $IBSS_N \longleftarrow IBSS(\delta_i.src, TOP_N)$
 // γ, code object in Γ with top SS.
19 $\gamma \longleftarrow argmax(SS(\delta_i.src, IBSS_N))$
 // Programmatic representation SS is γ's Combined IBSS/Attr SS scores.
20 $PRSS \longleftarrow max(SS(IBSS_\gamma, Attr_\gamma))$
21 $\Omega \longleftarrow \Omega \cup (\gamma, \delta)$ // Add δ, γ to Mapping Set.
 // Remove δ, γ from Δ, Γ.
22 $\Gamma \longleftarrow \Gamma \setminus \{\gamma\}$
23 $\Delta \longleftarrow \Delta \setminus \{\delta_i\}$
24 **while** $\Delta \neq \emptyset \parallel \Gamma \neq \emptyset$;
25 $UR_{SS} = Combine(MR_{SS},\ Attr_{SS},\ PR_{SS})$
26 **if** $UR_{SS} > max(MM)$ **then**
27 Matched Model $UR \longleftarrow UR_i$
 // UR_{SS} added to MM scores.
28 $MM \longleftarrow MM \cup UR_{SS}$
29 **end**
30 **end**

To address these issues, ZEN employs a novel model attribution algorithm that combines similarity scores (SS) calculated for the mathematical representation, attributes, and byte-codes between the recovered model and models in the base model library.

ZEN's model attribution is shown in Algorithm 1. First, each code object in the recovered DL system (M) is added to set Δ. ZEN begins attribution by comparing code objects in Δ to ones stored in the base model library (Line 3 - Line 7). ZEN first filters based on code object byte-code, as a 100% match on a byte-code comparison between a code object in Δ to one in the base model library indicates an identical function (Line 4). Any unified representation from the base model library containing

an identical function to a function in M is stored in an ordered list URO (Line 5).

For each unified representation UR_i in URO, ZEN first calculates a mathematical representation similarity score of UR_i to M (Line 10). ZEN does this by calculating the normalized differences among the parameters, layer shapes, and count of layer types. ZEN then employs a graph-based matching approach (our approach utilizes spectral graph matching [61], but other graph-matching algorithms are also viable) to get a graph similarity score. ZEN computes the mathematical representation similarity score (MR_{SS}) by taking a weighted average of the graph and layer attribute similarity scores. This average is weighted toward the model's graph similarity, as the overall architecture is a more stable identifier than layer-specific attributes (e.g., parameter counts, shapes), which can change dramatically with minor model variations like increasing a layer's width.

Then, each code object in UR_i is added to the matching set: Γ (Line 11). For each member in Γ, a decompilation of its code object's byte-code is made and attached to that member. ZEN then selects a single member δ from Δ as a current match candidate. A comparison is made by ZEN between the attributes (strings, variable names, variable type strings) of δ and the attributes of all code objects in Γ, producing an attribute similarity score (Attr. SS) (% of matching attributes).

ZEN identifies a set of code objects (Line 17) in Γ with the highest attribute similarity scores (TOP_N) relative to δ based on an element-wise distance calculation. This distance is calculated among the number of function arguments, local variables, and function calls in the code object's byte-code. In our evaluation, we use an N of 3 since we found that most DL systems did not share more than three functions with exact matching attribute similarity score. As multiple functions can have the same attribute counts, ZEN constructs TOP_N as a set where each code object in the set has the highest, but also equivalent, attribute similarity score to δ. To find a matching code object to δ, ZEN compares the byte-codes of δ to the byte-codes of the code objects in TOP_n.

Unfortunately, different framework versions (i.e., Python 3.7 vs. 3.8) may have changes in opcodes, resulting in byte-code variance [34]. This means that given some source code, the byte-code for that source code may vary from deployment to deployment depending on the framework version. However, we observed that though byte-codes may be different between a function and a modified version of that function, each function's corresponding source code can still be very similar.

Consequently, ZEN compares decompiled byte-codes for each code object in TOP_N to δ's decompiled byte-code with fuzzy string comparison, generating an implied byte-code similarity score (IB_{SS}) for each code object of TOP_N (Line 18). For each code object in TOP_N the attribute similarity score and IB similarity score are averaged into a programmatic representation similarity score (PR_{SS} in Line 20). For the code object of TOP_N, γ (Line 19), that has the highest programmatic representation similarity score, a mapping between code object δ and code object γ is stored in a *bipartite graph*: Ω (Line 21), indicating a matched function. Once added to Ω, γ and δ are removed from Γ and Δ, respectively (Line 22-Line 23) and this process continues until either Δ or Γ is empty.

Finally, a unified representation similarity score (UR_{SS}) is generated based on the combined mathematical and programmatic representation similarity scores of matched code objects (Line 25). If the current unified representation similarity score exceeds the current maximum unified representation similarity score (Line 27) in MM (set of current unified representation similarity score scores), the current attributed model is set to UR_i (Line 27) and added to the set of unified representation scores MM.

Algorithm 2: ZEN's Model Differential Analysis.

Input: UR: MM, Map Set: Ω, Unmapped CO Set: Δ
Output: CO Differences: D, Unique CO Set: Υ, Changed CO Set: ϵ

```
1  Υ ⟵ ∅
   // Corruption Check On Unique Code Objects
2  for δ ∈ Δ do
       // Ignore δ if Corrupt Attribute is Found
3      for Attr_i ∈ δ do
4          if Corrupt(Attr_i) then
5              δ.inv = True
6              break
7          end
8      end
9      if δ.inv == False then
10         Υ ⟵ Υ ∪ δ // Add Valid, Unique COs to Υ
11     end
12 end
13 ϵ ⟵ ∅
   // Process Differences of Changed Code Objects
14 for P_i ∈ Ω do
       // Initialize Changes Set for Current
          Mapping
15     c_i = ∅
16     for Attr_j ∈ P_i do
           // Ignore P_i if Corrupt Attribute is
              Found
17         if Corrupt(Attr_j.rec) then
18             P_i.inv = True
19             break
20         end
           // Include Nested COs in Mapping
21         if CO_Nest ∈ Attr_j.rec then
22             P_i ⟵ Update(P_i, CO_Nest)
23         end
           // Compute Change in Attribute
24         c_i ⟵ c_i ∪ Difference(Attr_j.base, Attr_j.rec)
25     end
26     if P_i.inv == False then
           // Change for each P_i Recorded in ϵ.
27         ϵ ⟵ ϵ ∪ c_i
28     end
29 end
```

C. Recovered Model Differential Analysis

With an identified base model, ZEN aims to find the differences between the base and recovered models to generate patches (in §IV-D) such that the recovered model can be reused in the base model's execution context $\hat{\mathcal{E}}$. However, conducting such differential analysis introduces technical challenges.

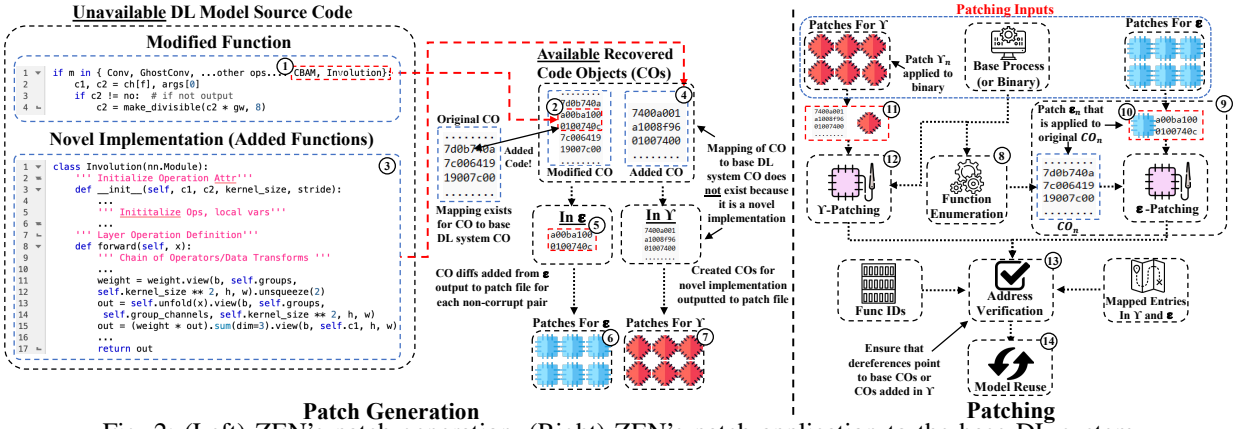

Fig. 2: (Left) ZEN's patch generation. (Right) ZEN's patch application to the base DL system.

First, the base and recovered models may have large discrepancies in counts of functions, classes, etc. This means that functions in the recovered system may have been added, changed, or even deleted relative to the base model. Even worse, recovered system functionality may be obfuscated (naming changes, code obfuscation, etc.), further muddying differential analysis. Second, even with all mappings in Ω between the base and recovered model's code objects produced in §IV-B1, the recovered DL system may introduce changes that do not correspond to anything in the base model (e.g., the addition or removal of classes/functions). These changes can be required to execute the recovered model (i.e., `Involution` class addition in §II-A). While model attribution produces a base model and estimates the execution context of the recovered model as $\hat{\mathcal{E}}$, differential analysis should output the changes that were made in the recovered DL system relative to the base model.

To address these challenges, ZEN applies a differential analysis algorithm, Algorithm 2, which can produce forensic evidence, as shown in §D. Note that Γ, post-attribution, contains the remaining code objects from the base model's unified representation that do not have a match to any code object in Δ. This implies that those functions are not used in the recovered model and can be ignored. However, Δ, post-attribution, contains the remaining code objects from the recovered model's unified representation that do not have a direct match to any code object in the base model's unified representation (this is the case when Γ is emptied first in Algorithm 1). This implies that connections in mapping set Ω are missing for the remaining code objects in Δ, meaning that instead of a unique change being applied to an existing code object in the base model's unified representation, an additional code object is implemented in the recovered model required for model reuse. Each member of Δ therefore corresponds to an additional code object (not in the base model) that needs to be added via patching into the base model. ZEN ensures that all attributes of each member of Δ have pointers to objects of known type and point to a valid region of memory (in the process heap) (Line 4). Members containing valid attributes are assigned to Υ (Line 10).

Next, for each mapped code object pair P_i in Ω, ZEN checks the attributes of P_i for corruptness (i.e., does a pointer point to an object within the memory mapping of the process and does the object have a valid Vtable pointer or type string). ZEN ignores all corrupt pairings for patching. For attributes with pointers to other code objects (i.e., helper functions), all nested code objects are recovered and included in modified pair mapping P_i (Line 21). ZEN records all changes in the attributes of the code object, such as argument counts, local variable counts, etc. (Line 24). Likewise, the differing byte-code is included in the modified pair. To conclude differential analysis, ZEN combines these changed elements into a single member, stored in ϵ (Line 27).

D. Model Patching

Referring back to the investigation in §II-A, Figure 2 shows the **unavailable** source code that was used in the recovered DL system. ZEN first takes the code objects from its differential analysis and generates a patch file corresponding to each set (changed code set ϵ and added code set Υ). For set ϵ (i.e., modified code in Figure 2 ①), corresponding to existing code objects in the base model that are changed in the recovered model, patches instead overwrite existing code objects at runtime (for interpreter-based DL systems). This means that for each pair in ϵ, the differences (⑤) are output directly to the file for patching the code object at runtime (⑥).For Υ, (i.e., code such as the added Involution code shown in ③), which contains novel implementations in the recovered model, a patch file is generated from ZEN-recovered code objects consistent with the system version of the recovered model's DL system (④). To do this, ZEN marshals each code object's byte-code along with its collected attributes, return address, local variables, etc. ZEN outputs this marshalled code object, along with its function ID (§IV-A), to the patch file (⑦).

Shown in Figure 2, given the patches generated and the base process, ZEN begins by enumerating the recovered DL system's code objects at runtime ⑧. For code objects found with an entry in ϵ, ZEN identifies the associated patch (⑩)

from the corresponding patch file and patches the attributes and byte-code of the base system code object such that it matches the recovered code object (9). To ensure that pointers in the modified code object dereference to the appropriate DL system function, ZEN first finds any function IDs (generated in §IV-A) referenced in that object's code (via call addresses). For each function ID found in the modified code object, ZEN replaces pointers to incorrect addresses within the code object to the correct addresses found in the function IDs list of the base implementation (or in ϵ or Υ, which contains the mappings for code objects between base and recovered system (13)). For interpreter-based systems this occurs at runtime, with ZEN patching existing code objects in memory. This is accomplished by iterating through all objects currently tracked in the Garbage Collector (GC) and filtering for code objects. ZEN then matches the function's `__name__` and `__module__` attributes against the function ID of the function to be patched. Once the target code object is located, the patch is applied by modifying its `__code__` attribute. The new code object is created by deserializing the marshalled byte-code from the patch file using Python's built-in marshal module. This effectively replaces the function's logic at runtime.

Now, for each added code object in Υ ((11)), ZEN patches a new code object into the base model's DL system ((12)). ZEN first unmarshalls all code object entries, in Υ, within the patch file. Similar to ZEN's processing of entries of ϵ, ZEN employs address verification ((13)) to ensure that all pointers in the code object's byte-code dereference to existing functions within the DL system programmatic representation. For incorrect addresses found in the code object's byte code, ZEN aims to identify whether the address corresponds to an existing entry in Υ or instead exists as a modified function in ϵ. If in neither, ZEN identifies the function via its function ID (from the base model's unified representation), and patches the address to the correct call address of the existing base model function. Then, for each unmarshalled code object, ZEN applies Python's `types.FunctionType` constructor to create a new function object from this code object and the global namespace of the target module. The newly created function is then injected into the target module's namespace using the `setattr()` built-in Python function. This makes the new function available to be called by other parts of the model's code. Finally, by enumerating objects in the GC, ZEN fixes incorrect addresses within newly added functions to correct entries in the GC based on either existing entries in Υ or ϵ. If in neither Υ nor ϵ, ZEN once again identifies the function via its function ID from the base system unified representation, and patches the address to the correct call address. Once this process is complete, the model will properly execute the uniquely implemented code recovered from the recovered DL system ((14)).

V. Evaluation

ZEN's algorithm is generic, but our prototype implementation consists of ∼5000 lines of Python code targeting the recovery and reuse of Python-based SOTA DL systems, a choice based on Python's popularity for DL system deployment. We built the base model library for ZEN by deploying foundational models, collecting a memory image for each at inference time (entire system physical memory via LiME [48]), and inputting each memory image to ZEN. The outputted unified model representation was inserted into the base model library.

A. Experimental Setup

ZEN was evaluated across 21 DL systems. For each of the 21 models tested, ZEN was provided only with the corresponding memory image and access to its base model library. Crucially, ZEN had no *a priori* knowledge of the true base model for any test case; each system was treated as entirely unknown, mirroring a real-world forensic scenario. We deployed seven base models alongside 14 customized models. These models are based on state-of-the-art research and are hosted on open-source AI Marketplaces (i.e., HuggingFace [2], Github [3]). For each base model, we found two customized models targeting the vision and language application domains. We denote the grouping of the base/customized models as a *model family*. Models were deployed on an Ubuntu 20.04 system with 64 GB of memory and an NVIDIA A6000 GPU. Similarly, ZEN was deployed on the same setup for model recovery, unified model representation synthesis, model attribution, and model reuse.

a) Model Selection and Diversity: Prior work [4] has suggested that proprietary systems often utilize open-source code when building their own proprietary DL systems (upward of 89% of participants relied heavily on open-source code). Based on this, we selected customized models that build upon existing open-source code/models. Our evaluation spans seven distinct model families enumerated in §V-A1.

Columns 1-3 in Table II highlight the architectural drift between base and customized models via the tensor and parameter counts of each model. We observed structural changes resulting in a difference of up to **200 tensors** between a customized model and its base (YoloV8-Gold [68]). On average, our selected customized models feature approximately 50 more tensors than their associated base models. Selected models also show high variance of model parameters. For example, some customized models significantly expand on their base, such as Llama 3, which increases the parameter count to 8.0B weights from its 6.7B-weight Llama 2 base. Conversely, other models are heavily pruned, like YoloV8-SPD, which contains just 3.5M weights, a fraction of the 25.8M in the original YoloV8.

Columns 7-11 of Table II show that practitioners frequently added new classes (an average of four per model) and, in some cases, performed substantial code refactoring by removing over 170 code objects from the original implementation (YoloV8-SPD [60]). Our dataset even includes the case of a complete rewrite of a model's codebase into a different programming style (Llama2-PT [76]), ensuring that ZEN is tested against a comprehensive suite of realistic modification scenarios. As shown in column 11 of Table II,

TABLE II: ZEN's Model Attribution, Patch Generation, and Model Reuse on Deployed Customized Models.

Deployed Model	Tensors	Weights	Base Model	Dataset	Attributed Model By ZEN	Identified Changes From Base Model[1]					Model Performance[2]		
						Code Objects		Classes	Funcs	Changes	Deployed	ONNX[3]	Post-ZEN
						Edits	Added						
YoloV5 [29]	177	7.2M	-	COCO [62]	YoloV5	0	0	13	334	0.0%	0.568	0.568	0.568
YoloV5-KD [63]	177	7.2M	YoloV5	COCO	YoloV5	6	9	13	341	4.4%	0.561	0.561	0.561
YoloV5-HIC [54]	230	9.3M	YoloV5	VisDrone [64]	YoloV5	3	8	22	342	3.2%	0.413	∅	0.413
YoloV7 [65]	288	37.2M	-	COCO	YoloV7	0	0	15	365	0.0%	0.69	0.69	0.69
YoloV7-C3 [66]	263	33.8M	YoloV7	VisDrone	YoloV7	13	27	21	388	10.9%	0.365	∅	0.365
YoloV7-GAM [67]	321	31.2M	YoloV7	VisDrone	YoloV7	13	27	22	388	9.0%	0.327	∅	0.327
YoloV8 [59]	167	25.8M	-	COCO	YoloV8	0	0	13	886	0.0%	0.631	0.631	0.631
YoloV8-Gold [68]	367	24.2M	YoloV8	VisDrone	YoloV8	156	125	28	750	37.5%	0.339	∅	0.339
YoloV8-SPD [60]	184	3.5M	YoloV8	VisDrone	YoloV8	150	256	15	715	56.8%	0.341	∅	0.341
Resnet [69]	62	11.2M	-	CIFAR10 [70]	Resnet	0	0	6	6	0.0%	92.34%	92.34%	92.34%
Resnet-Ghost [71]	127	56.2M	Resnet	CIFAR10	Resnet	0	5	11	10	42.9%	93.12%	∅	93.12%
Resnet-SE [72]	189	5.7M	Resnet	CIFAR10	Resnet	1	4	8	8	62.5%	92.88%	∅	92.88%
MobileNetV2 [73]	158	2.2M	-	CIFAR10	MobileNetV2	0	0	7	30	0.0%	79.4%	79.4%	79.4%
MobileNetV2-Ghost [71]	281	4.7M	MobileNetV2	CIFAR10	MobileNetV2	0	11	10	39	28.2%	83.6%	∅	83.6%
MobileNetV3 [74]	174	5.5M	MobileNetV2	CIFAR10	MobileNetV2	3	12	13	18	83.3%	82.2%	∅	82.2%
Llama 2 [75]	291	6.7B	-	Undisclosed[4]	Llama 2	0	0	8	23	0.0%	100.0%	100.0%	100.0%
Llama-PT [76]	291	6.7B	Llama 2	Undisclosed	Llama 2	5	7	7	20	25.0%	100.0%	∅	100.0%
Llama 3 [21]	291	8.0B	Llama 2	Undisclosed	Llama 2	10	10	8	27	74.0%	100.0%	∅	100.0%
nanoGPT [77]	40	10.7M	-	OpenWebText [78]	nanoGPT	0	0	10	20	0.0	3.16	3.16	3.14
nanoGPT-LORA [31]	101	134.7M	nanoGPT	OpenWebText	nanoGPT	2	14	10	29	55.1%	3.22	∅	3.21
nanoGPT-RWKV [79]	76	10.8M	nanoGPT	OpenWebText	nanoGPT	0	6	11	20	30.0%	3.14	∅	3.15

1: Classes, functions, code objects, modules patched. Changes: percentage of the number of programmatic representation functions changed in patching.
2: Accuracies are presented as mAP for Yolo models, percentage correct on a test dataset for Resnets and Mobilenets, and validation loss for Llamas and nanoGPTs.
3: Utilizing an ONNX representation based on recovered model architecture, weights, tensors, etc is a strawman solution for model redeployment (§II-A).
4: Undisclosed dataset by model creator. Pre-trained weights are used for the experiment.

our evaluated models varied from their base model from 3.2% (YoloV5-HIC [54]) to upward of 83.3% (MobileNetV3 [74]).

1) Model Setup: The models and datasets used in our experiments are shown in Columns 1 and 5 of Table II. The training of each model directly followed the protocols in each associated paper/documentation, which can be found in the cited work.

a) Vision Models: To evaluate vision models we used five model types in our base model library. Models were trained on the VisDrone [64] (drone based detection/tracking), COCO [62] and CIFAR10 [70] (10 class, 32x32 images) datasets. Constituting the YoloV5 [29] model family, we deployed YoloV5, YoloV5-KD [63], and YoloV5-HIC [54], which were trained on COCO, COCO, and VisDrone, respectively. The YoloV8 [59] model family (YoloV8, YoloV8-SPD [60], and YoloV8-Gold [68]), was trained on COCO, VisDrone, and VisDrone respectively, and was deployed. Finally, the YoloV7 [65] model family (YoloV7, YoloV7-C3 [66], and YoloV7-GAM [67]) was trained on COCO, VisDrone, and VisDrone, respectively.

For image classification, we deployed Resnet [69] and MobileNetV2 [73] base models trained on the CIFAR10 dataset. The customized implementations of Resnet (Resnet-Ghost [71] and Resnet-SE [72]) and MobileNetV2 (MobileNetV2-Ghost [71] and MobileNetV3 [74]) were also trained on the CIFAR10.

b) Language Models: For language models, we deployed the Llama2 [75] and nanoGPT [77] models, which were respectively trained on the proprietary WebText dataset (unreleased by Meta) and the OpenWebText [78] (open-source estimation of WebText) dataset. Llama2-PT [76] (PyTorch-only implementation of Llama2) and Llama3 [21] (SOTA Llama2 successor) were used as the custom DL systems based on Llama2. Since the dataset and training protocol are undisclosed for Llama models, we deployed each model with a set of pre-trained model weights. Lacking the original dataset, we evaluated the performance of Llama models using 100 randomly selected OpenWebText [78] samples. We performed manual verification of the model output, looking for reasonable text generation and contextual awareness as evidence that model reuse was successful. We also used base model nanoGPT [77], an implementation of GPT2 [80], which was trained on the OpenWebText [78] dataset. Custom implementations LoRA [31] and RWKV [79] were also trained on the OpenWebText dataset.

B. Customized Model Reuse

ZEN was able to recover, attribute, and reuse 21/21 of the deployed models (100%). Table II describes ZEN's end-to-end operation. Given a memory image containing a DL system, ZEN first recovers the tensors and weights of each model (Columns 2-3). Then, ZEN attributes the unknown model to a model within its base model library (Column 4). We observe that in all model deployments (21/21) ZEN correctly attributed the unknown customized model to the ground truth base model (Column 6). Regardless of dataset used for training (i.e., COCO vs. VisDrone in all Yolo models), model attribution is **not** affected. This is because, while essential to use-case-specific model reuse, model weight **values** are not a part of the MR.

a) Customization From Base Model: Upon model attribution, ZEN identifies customizations for the base model to enable customized model reuse. Columns 7-8 show the number of differences identified by ZEN for code objects modified and added (ϵ and Υ in §IV-D) to the base model to enable reuse of the customized model. Shown in Column 9-10 are the classes and functions defining the programmatic

representation of the deployed model. While having upwards of 886 functions (YoloV8) and 28 classes (YoloV8-Gold), the relative difference in the custom models and base models (highlighted by changes (%) shown in Column 11) depends on the size of the base DL system. The minimum number of changes identified by ZEN was five changes (Resnet family). As the Resnet base model was implemented using standard DL framework built-in layers (i.e. PyTorch Conv layers), its customized models thereby contain little user-defined code modifying existing functionality (only one modified code object). Instead, the changes for small model families (Resnet, MobileNetV2, and nanoGPT) often introduce code objects, as shown in Column 8. Similarly, even though the number of changes identified for such smaller DL systems is less than the number identified for larger DL systems (Yolo families, Llama), the relative difference (Column 11) between the base and custom systems can still remain large (i.e., 62.5% difference for Resnet-SE).

The maximum number of changes identified by ZEN was 406 for YoloV8-SPD (150 and 256 modified/added code objects). As the base model (and other Yolo family models) were built on the Ultralytics [58] code base, the size of the programmatic representation (886 functions in Column 8) was larger than for model families with smaller code bases (e.g., max of 39 in MobileNetV2-Ghost). For models such as YoloV7, we notice that though the number of changes found for such systems is larger (40 found for YoloV7-GAM) than the number generated for smaller DL systems (five found for Resnet), the relative difference (Column 11) is small (i.e., 62.5% difference for Resnet-SE vs. 9.0% for YoloV7-GAM). This is because the base model's programmatic representation size is much larger than the change count implemented by the DL practitioner in the customized model.

We found that in 19/21 (90.5%) models (excluding YoloV8-Gold and Llama 3) changes were overwhelmingly introduced as **added** code objects. This highlights our insight that DL-practitioners often rely on base models to build new functionality. We also observe that for smaller model families, changes were almost exclusively introduced as added code objects (87.5%, or 32/36, for the Resnet and MobileNetV2 families). As the base DL system code bases were small, or using very primitive DL architectures (Resnet), functionality changes were primarily made through added code.

b) Model Reuse: Upon successful difference identification, patch generation, and application to ZEN attributed base models, we measure the performance of the models on their respective datasets. We show that upon reuse, the recovered model will have the same performance as during deployment (ground truth performance in Column 12).

Prior work has considered an ONNX representation as an end goal of model recovery and reuse. Column 13 shows where this approach fails. As discussed in §II, an ONNX representation relies on knowledge of the model's architecture, connections, weights, *and implementation*. This knowledge is readily available for base model implementations derived from AI marketplaces. However, for customized models

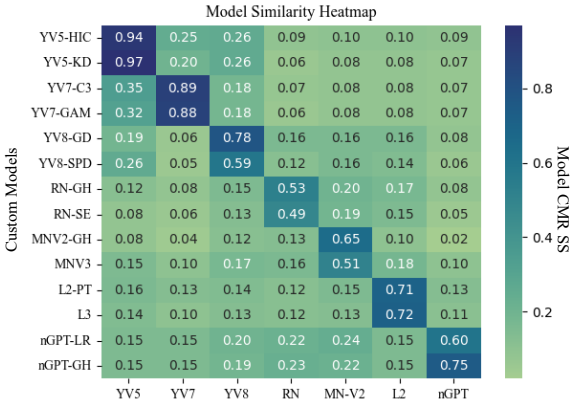

Fig. 3: A heatmap depicting high association of customized models to their respective base model.

implementing changes upon the base model, we found that in 13/14 models (excluding YoloV5-KD), changes were introduced primarily as added code objects affecting the model's programmatic representation. Without reverse engineering and manually determining this set of added (and modified) code objects within the programmatic representation of the customized model (an average of ∼226 functions), even getting the ONNX representation remains an impossible task, impeding practical reuse (indicated in Column 13 by ∅).

Shown in Column 14 is ZEN's post-patching model performance. We observe that in all instances the reused model performance mirrors its pre-recovery deployment performance (on the same test data), indicating successful model reuse in 21/21 models (100%). Note that for the nanoGPT family, validation loss was used as the performance metric. As a result, upon model reuse and training resumption, nanoGPT loss was different than during deployment (a small difference of up to 0.02). Interestingly, for YoloV5-KD, a customized YoloV5 model utilizing knowledge distillation [63], reuse in an ONNX representation is possible. As YoloV5-KD does not make changes directly to the model implementation, but instead introduces novel implementations *surrounding the model*, the mathematical/ programmatic representations almost match those of the base model (§V-C).

C. Model Attribution

Figure 3 is a heat map depicting the unified representation similarity scores between customized and base models, highlighting ZEN's model attribution. A higher similarity score is indicated by a darker color in the heat map, whereas a lower one is indicated by a lighter color. An indication that ZEN successfully attributed each customized model to its associated base model, we observe that the highest similarity for each model lies along the diagonal of the heat map. The average similarity score across this diagonal (ZEN attributed models) was 0.72, whereas adjacent diagonals had average scores of 0.16.

The maximum similarity (0.97) was seen for the YoloV5-KD model customization, which can be explained by its implementation not changing the model operation code but

TABLE III: ZEN's Output of Similarity Scores (MR and PR) Customized Models With Respect to Each Model in the BML.

CM[1]	YV5		YV7		YV8		RN		MN-V2		nGPT		L2	
	MR	**PR**	**MR**	**PR**	**MR**	**PR**	**MR**	**PR**	**MR**	**PR**	**MR**	**PR**	**MR**	**PR**
YV5-KD	**1.0**	**0.96**	0.04	0.25	0.46	0.19	0.26	0.0	0.32	0.0	0.33	0.0	0.25	0.01
YV5-HIC	**0.82**	**0.98**	0.21	0.26	0.45	0.19	0.37	0.0	0.39	0.0	0.42	0.0	0.32	0.01
YV7-C3	0.6	0.27	**0.73**	**0.94**	0.45	0.09	0.27	0.0	0.27	0.01	0.31	0.01	0.26	0.01
YV7-GAM	0.47	0.27	**0.68**	**0.94**	0.43	0.09	0.23	0.0	0.28	0.01	0.3	0.01	0.25	0.01
YV8-GD	0.47	0.1	0.11	0.05	**0.54**	**0.86**	0.32	0.1	0.35	0.1	0.36	0.1	0.3	0.0
YV8-SPD	0.58	0.15	0.02	0.06	**0.46**	**0.63**	0.18	0.1	0.36	0.1	0.25	0.1	0.25	0.0
RN-GH	0.46	0.0	0.3	0.0	0.3	0.1	**0.63**	**0.5**	0.5	0.1	0.38	0.1	0.3	0.0
RN-SE	0.33	0.0	0.25	0.0	0.21	0.1	**0.47**	**0.5**	0.47	0.1	0.29	0.1	0.21	0.0
MNV2-GH	0.23	0.03	0.06	0.03	0.06	0.14	0.22	0.1	**0.32**	**0.76**	0.06	0.11	0.06	0.01
MNV3	0.46	0.05	0.25	0.05	0.25	0.14	0.36	0.1	**0.51**	**0.51**	0.29	0.15	0.25	0.05
nGPT-RK	0.45	0.05	0.45	0.05	0.45	0.1	0.61	0.1	0.45	0.15	**0.71**	**0.76**	0.45	0.05
nGPT-LR	0.46	0.05	0.44	0.05	0.5	0.1	0.58	0.1	0.5	0.15	**0.55**	**0.61**	0.44	0.05
L3	0.54	0.03	0.43	0.03	0.54	0.0	0.48	0.0	0.53	0.02	0.46	0.02	**0.99**	**0.62**
L2-PT	0.53	0.01	0.42	0.0	0.52	0.0	0.46	0.0	0.52	0.0	0.44	0.0	**1.0**	**0.62**

1: The customized model that is recovered by ZEN.
YV7 → YoloV7, YV5 → YoloV5, YV8 → YoloV8, RN → Resnet,
MNV2 → MobileNetV2 nGPT → nanoGPT, L3 → Llama 3, L2 → Llama 2

rather the code surrounding the model's implementation. The minimum similarity is seen for Resnet-SE (0.49) because the size of its base model programmatic representation is small, meaning that even small changes (five patches) are enough to significantly affect model similarity. Inversely, models with larger base model programmatic representation sizes (i.e., Yolo model families and Llama2) had higher similarity to their base models. We also observe that models extending the same underlying framework and similar operation types tend to be more similar (clumping of darker colors for models in the same family). For example, for YoloV7-C3, the base Yolo models that were *not* YoloV7 had up to 28% (for YoloV5) more similarity than other model families (i.e., Resnet).

Table III presents the similarities between each customized model and base model broken down into mathematical and programmatic representation similarities. Shown across the diagonal of the table, are the similarities for each customized model relative to their base model. For models in the same family, the similarities are highest (e.g., YoloV5-KD and YoloV5-HIC have 1.0 & 0.82 mathematical representation and 0.96 & 0.98 programmatic representation similarities). We observe that whereas the heatmap showed overall similarity between customized and base models, Table III shows the contributions of the mathematical and programmatic representations, explaining *why* models were similar/different.

Looking at the Llama model family, we see that while the code was significantly changed in customized versions (Llama2-PT and Llama3 both having a programmatic representation similarity of 0.62), the mathematical representation for each was almost exactly the same (0.99 for Llama3 and 1.0 for Llama2-PT) as the base model. This means that even if the model architecture is mostly unchanged, the customized DL system implementation can be significantly different from the base system, resulting in different model performances. Universally, we notice that regardless of mathematical representation changes, the programmatic representation was affected by new code added to the customized model. For models with large changes in mathematical representation (i.e., MobileNetV2 and MobileNetV2-Ghost with a similarity of 0.32), the programmatic representation similarity can still remain high (0.76 for MobileNetV2-Ghost). For attribution, this indicates that code differences have the highest impact on model attribution, which matches the intuition argued in §II. Discussed in §A, mathematical and programmatic representation similarities can be further broken down into more granular similarity measurements.

a) Interpretation by a Forensic Investigator: The similarity scores produced by ZEN provide quantitative evidence for ZEN's attribution. For example, when comparing YV5-HIC (Table III Row 2) to *incorrect* base models, ZEN outputs an average similarity score of only 0.15. However, when compared to YV5 (the correct base model), YV5-HIC shows high mathematical and programmatic similarity scores of 0.82 and 0.98, respectively. The similarity scores are fine-grained enough to distinguish between models from the same family. For example, the most similar model to YV5 (other than its base model) is YV7-C3, with a similarity score of only 0.35 shown in Figure 3.

D. Case Study: License Enforcement

Models used in production heavily rely on open-source foundational models for high accuracy, use-case-specific, deployments [1]. Such foundational models are offered to individuals and organizations by large AI research companies [2], [58], [81] under a variety of licenses. Notably, such platforms provide **serverless** frameworks and SOTA models for direct non-commercial use *without a proprietary license* (e.g., copyleft licenses such as AGPL [82]). This directly enables the use of SOTA models offline for a range of deployment scenarios. This enables further research and provides users unfettered access to SOTA models for personal projects. Inversely, the same platforms offer a variety of proprietary licenses that enable the commercial use of their framework/models for other companies.

Unfortunately, this introduces problems related to the **enforcement** of such licenses [83]. First, infringements upon such licenses are often unknown to license distributors (as companies may rename models and framework data structures and even run models serverless, with no API access necessary). Second, even if a company suspects license infringement, proving that its model/framework was indeed infringed upon is challenging (DL system can be modified, obfuscated, encrypted etc.).

a) Investigation with ZEN: YoloV8 is a SOTA vision model developed by Ultralytics. Imagine a forensic scenario where Ultralytics suspects that a new serverless DL system infringes upon its YoloV8 license. To evaluate such a case, we used YoloV10 [22] (released May 2024, developed by Tsinghua University) as the "suspect model." Note that YoloV10 is a legitimate extension to YoloV8 but implements significant variation upon YoloV8's code base and model architecture.

Table IV highlights such an investigation of a suspect DL system (utilizing a SOTA YoloV10 [22] implementation).

TABLE IV: Investigating a Suspect Model.

ZEN Input	Suspect Memory Image
Unified Representation:	**Tensors:** 213 **Layers:** 24 **Weights:** 2.7M **Functions:** 869 **Modules:** 74
Incriminating Artifacts:	**Functions:** ['load_model', ...] **Modules:** ['ultralytics.utils', ...] **Classes:** ['Concat','C2fCIB',...]
Model Attribution[1]:	**Similarity Score Analysis:** Mathematical Representation: 0.34 Programmatic Representation: 0.92 Unified Representation: 0.77 **Base Model ID:** YoloV8 [59]

Given a memory image of the suspect DL system, ZEN first synthesizes a unified representation comprising 213 tensors, 24 layers, 2.7M weights, 869 functions, and 74 modules. ZEN then outputs recovered artifacts (i.e., code objects, classes, etc.) found in the DL system. From this, it can be seen that Ultralytics modules are present in the DL system and that the suspect system contains classes/functions matching those in Yolo family DL systems. This is further confirmed by ZEN's model attribution, where it can be seen that the model was directly attributed to the YoloV8 base model with mathematical, programmatic, and unified representation similarity scores of 0.34, 0.92, and 0.77, respectively. These findings can be presented by the investigator in court to demonstrate that the suspect DL system reuses Ultralytics (YoloV8 model owner) framework code and to what degree (e.g., what code objects were reused).

VI. Discussion

A. Extending ZEN to C/C++

While our prototype of ZEN is implemented for Python-based DL frameworks, its core methodology is conceptually extensible to compiled C/C++ systems with dedicated engineering effort. First, ZEN's methodology for unified representation synthesis (§IV-A) still holds, as all DL models inherently possess a computation graph, layer implementations with associated weights, and executable code. Engineering effort would focus on adapting ZEN's analysis to identify and parse the specific in-memory data structures of compiled C++ frameworks, rather than Python-based DL frameworks. Similarly, the core logic for attribution and differential analysis, which compares these unified representations, would remain conceptually the same.

Unlike the runtime patching used for Python, reconstructing a C/C++ model could also be done pre-deployment. ZEN would instead need to patch the compiled binary before execution of the binary. Although this requires a different set of tools and techniques, the feasibility of such an approach is well-supported by a robust body of existing research in binary analysis [84]–[86], code similarity detection [87]–[90], and binary patching [91]–[94].

B. Extending ZEN's Base Model Library

Extending the base model library with new models is an automated process designed to require minimal effort from an investigator. To add a new model, an investigator needs to provide ZEN with a memory dump of the known model during deployment along with a ground truth model name. From these inputs, ZEN applies the methodology from §IV-A. It extracts the model's computation graph and layer attributes to create the mathematical representation and recovers the underlying bytecode and function metadata to create the programmatic representation. ZEN combines these into a unified model representation following §IV-A and stores it in the base model library under its ground truth model name (investigator input). This makes the new model immediately available as a candidate for all future attribution and comparison tasks by ZEN.

C. Extending ZEN to New DL Frameworks

ZEN was prototyped to handle DL models deployed in the most common Python-based DL frameworks [95]. However, in the case that the target DL system utilizes a new DL framework, applying ZEN would require preliminary reverse engineering. Specifically, they would need to extend ZEN's unified representation recovery (§IV-A) to support the new framework. An investigator would first need to reverse engineer the data structures that represent models in memory. Then, the investigator would need to update the mathematical and programmatic representations used by ZEN to recognize and interpret the new framework's in-memory layouts for tensors, computation graphs, and code objects. Once these changes are made, the subsequent stages of ZEN (i.e., model attribution and differential analysis) would work as is.

VII. Related Work

Existing model recovery tools [14]–[16], [23], [24], while recovering the DL model's mathematical representation, fail to consider customized layers and operators.

Shown in Table V in §A, methodologies prior to ZEN are unable to recover (or can only partially recover) SOTA DL models. First, techniques such as LibSteal [24], DnD [14], BTD [15], and MXR [23] are unable to recover DL models from encrypted binaries, as they employ static analysis-based methodologies. MXT [23], AIP [16], and our work, ZEN, employ dynamic analysis- and memory forensics-based approaches, respectively, enabling recovery of DL models from encrypted binaries. Second, while all prior work can recover the DL Model's mathematical representation (besides LibSteal), none are able to recover the exact operators required for DL model inference. Prior work instead opts to infer **common** operator definitions via data structure [16] and loop analysis [14], learning-based approaches [15], or they ignore operator recovery entirely [23], [24]. The failure to recover operator definitions makes it infeasible to recover custom layers used in SOTA or proprietary DL models. ZEN instead recovers a unified representation of the model, enabling the recovery of customized layers and operator types. Ultimately, related work *assumes* that re-execution and reuse of the model are possible, though we find that the factors discussed in this paper severely impede reuse without unified model representation recovery and attribution.

VIII. Conclusion

ZEN is the first system aiming to fill an existing gap in model recovery SOTA work. ZEN synthesizes a unified model representation of a deployed DL system, attributes it to a base implementation, and applies generated patches to enable model reuse. We evaluated ZEN on 21 deployed DL models, and successfully recovered, attributed, and reused 21/21 models.

References

[1] A. Ho, *Proprietary ai models are dead. long live proprietary ai models*, [Accessed: 2023-11-16]. [Online]. Available: https://thenewstack.io/proprietary-ai-models-are-dead-long-live-proprietary-ai-models/.

[2] Hugging Face, https://huggingface.co/, [Accessed: 2023-10-06].

[3] GitHub, *GitHub*, 2024. [Online]. Available: https://github.com.

[4] K. Grosse, L. Bieringer, T. R. Besold, and A. M. Alahi, "Towards more practical threat models in artificial intelligence security," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[5] 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 *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2019.

[6] J. Guo, A. Li, and C. Liu, "AEVA: Black-box backdoor detection using adversarial extreme value analysis," in *Proceedings of the 10th International Conference on Learning Representations (ICLR)*, Virtual Conference, Apr. 2022.

[7] 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 *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[8] 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.

[9] E. Tjoa and C. Guan, "A survey on explainable artificial intelligence (xai): Towards medical xai," *IEEE Transactions on Neural Networks and Learning Systems*,

[10] Y. Jiang, Y. Gao, C. Zhou, H. Hu, A. Fu, and W. Susilo, "Intellectual property protection for deep learning model and dataset intelligence," *arXiv preprint arXiv:2411.05051*, 2024. [Online]. Available: https://arxiv.org/abs/2411.05051.

[11] *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].

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

[13] *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].

[14] R. Wu, T. Kim, D. (Tian, A. Bianchi, and D. Xu, "DnD: A Cross-Architecture deep neural network decompiler," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[15] Z. Liu, Y. Yuan, S. Wang, X. Xie, and L. Ma, "Decompiling x86 deep neural network executables," in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[16] D. Oygenblik, C. Yagemann, J. Zhang, A. Mastali, J. Park, and B. Saltaformaggio, "AI Psychiatry: Forensic Investigation of Deep Learning Networks in Memory Images," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

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

[18] 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.

[19] 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.

[20] K. Liu, B. Dolan-Gavitt, and S. Garg, "Fine-pruning: Defending against backdooring attacks on deep neural networks," pp. 273–294, Sep. 2018. DOI: 10.1007/978-3-030-00470-5_13.

[21] Meta, *Build the future of ai with meta llama 3*, [Accessed: 2024-07-11]. [Online]. Available: https://llama.meta.com/llama3.

[22] A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin, J. Han, and G. Ding, "Yolov10: Real-time end-to-end object detection," *2405.14458*, 2024. [Online]. Available: https://arxiv.org/abs/2405.14458.

[23] 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 *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.

[24] J. Zhang, P. Wang, and D. Wu, "Libsteal: Model extraction attack towards deep learning compilers by reversing dnn binary library," *International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE - Proceedings*, pp. 283–292, 2023. DOI: 10.5220/0011754900003464.

[25] L. P. Osco, J. Marcato Junior, A. P. Marques Ramos, L. A. de Castro Jorge, S. N. Fatholahi, and J. de Andrade Silva, "A review on deep learning in uav remote sensing," *International Journal of Applied Earth Observation and Geoinformation*, vol. 102, 2021. DOI: https://doi.org/10.1016/j.jag.2021.102456. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S030324342100163X.

[26] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, and E. Brynjolfsson, "On the opportunities and risks of foundation models," *2108.07258*, 2022. [Online]. Available: https://arxiv.org/abs/2108.07258.

[27] A. Shafahi, W. R. Huang, M. Najibi, O. Suciu, C. Studer, T. Dumitras, and T. Goldstein, "Poison frogs! targeted clean-label poisoning attacks on neural networks," in *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS)*, Montreal, Canada, Dec. 2018.

[28] 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.

[29] G. Jocher, *YOLOv5 by Ultralytics*, 2020. [Online]. Available: https://github.com/ultralytics/yolov5.

[30] FairScale authors, *Fairscale: A general purpose modular pytorch library for high performance and large scale training*, https://github.com/facebookresearch/fairscale, 2021.

[31] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *Proceedings of the 10th International Conference on Learning Representations (ICLR)*, Virtual Conference, Apr. 2022.

[32] Hexrays, *IDA Pro*, 2024. [Online]. Available: https://hex-rays.com/ida-pro/.

[33] NSA, *Ghidra Software Reverse Engineering Framework*, 2024. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra.

[34] R. Bernstein, *python-uncompyle6*, 2024. [Online]. Available: https://github.com/rocky/python-uncompyle6.

[35] L. Dramko, J. Lacomis, E. J. Schwartz, B. Vasilescu, and C. Le Goues, "A taxonomy of c decompiler fidelity issues," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[36] Z. Liu and S. Wang, "How far we have come: Testing decompilation correctness of c decompilers," in *Proceedings of the 2020 International Symposium on Software Testing and Analysis (ISSTA)*, Virtual Conference, Jul. 2020.

[37] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv:1412.6572*, 2015. [Online]. Available: https://arxiv.org/abs/1412.6572.

[38] W. Song, H. Yin, C. Liu, and D. Song, "Deepmem: Learning graph neural network models for fast and robust memory forensic analysis," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[39] A. Oliveri, M. Dell'Amico, and D. Balzarotti, "An os-agnostic approach to memory forensics.," in *Proceedings of the 2023 Annual*

Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2023.

[40] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Guitar: Piecing together android app guis from memory images," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.

[41] 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 *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[42] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[43] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[44] 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 *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[45] 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.

[46] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Vcr: App-agnostic recovery of photographic evidence from android device memory images," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.

[47] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "Dscrete: Automatic rendering of forensic information from memory images via application logic reuse," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[48] 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.

[49] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "Trustdump: Reliable memory acquisition on smartphones."

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

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

[52] 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.

[53] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, Vancouver, Canada, Dec. 2019.

[54] S. Tang, S. Zhang, and Y. Fang, "Hic-yolov5: Improved yolov5 for small object detection," *2309.16393*, 2023. [Online]. Available: https://arxiv.org/abs/2309.16393.

[55] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, and J. Sun, "Repvgg: Making vgg-style convnets great again," in *Proceedings of the 38th IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Virtual Conference, Jun. 2021.

[56] P. S. Foundation, *Code objects*, [Accessed: 2023-11-16]. [Online]. Available: https://docs.python.org/3/c-api/code.html.

[57] *Memory smear*, https://capsicumgroup.com/glossary/memory-smear/, [Accessed: 2024-08-12].

[58] G. Jocher, A. Chaurasia, and J. Qiu, *Ultralytics YOLO*, 2023. [Online]. Available: https://github.com/ultralytics/ultralytics.

[59] G. Jocher, A. Chaurasia, and J. Qiu, *Ultralytics YOLO*, 2023. [Online]. Available: https://github.com/ultralytics/ultralytics.

[60] R. Sunkara and T. Luo, "No more strided convolutions or pooling: A new cnn building block for low-resolution images and small objects," *2208.03641*, 2022. [Online]. Available: https://arxiv.org/abs/2208.03641.

[61] M. Leordeanu and M. Hebert, "A spectral technique for correspondence problems using pairwise constraints," *ICCV '05*, 2005. DOI: 10 . 1109 / ICCV . 2005 . 20. [Online]. Available: https://doi.org/10.1109/ICCV.2005.20.

[62] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," *1405.0312*, 2015. [Online]. Available: https://arxiv.org/abs/1405.0312.

[63] T. Wang, L. Yuan, X. Zhang, and J. Feng, "Distilling object detectors with fine-grained feature imitation," in *Proceedings of the 36th IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Long Beach, CA, Jun. 2019.

[64] P. Zhu, L. Wen, D. Du, X. Bian, H. Fan, Q. Hu, and H. Ling, "Detection and tracking meet drones challenge," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, pp. 7380–7399, 2021. DOI: 10.48550/arXiv.2001.06303.

[65] 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 *Proceedings of the 40th IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Vancouver, Canada, Jun. 2023.

[66] H. Park, Y. Yoo, G. Seo, D. Han, S. Yun, and N. Kwak, "C3: Concentrated-comprehensive convolution and its application to semantic segmentation," *arXiv:1812.04920*, 2018. [Online]. Available: https://arxiv.org/abs/1812.04920.

[67] Y. Liu, Z. Shao, and N. Hoffmann, "Global attention mechanism: Retain information to enhance channel-spatial interactions," *arXiv:2112.05561*, 2021. [Online]. Available: https://arxiv.org/abs/2112.05561.

[68] C. Wang, W. He, Y. Nie, J. Guo, C. Liu, K. Han, and Y. Wang, "Gold-yolo: Efficient object detector via gather-and-distribute mechanism."

[69] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the 33rd IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, Nevada, Jun. 2016.

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

[71] C. Chen, Z. Guo, H. Zeng, P. Xiong, and J. Dong, "Repghost: A hardware-efficient ghost module via re-parameterization," *arXiv:2211.06088*, 2022. [Online]. Available: https://arxiv.org/abs/2211.06088.

[72] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the 35th IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Salt Lake City, Utah, Jun. 2018.

[73] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the 35th IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Salt Lake City, Utah, Jun. 2018. DOI: 10.1109/CVPR.2018.00474.

[74] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, *et al.*, "Searching for MobileNetV3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, Seoul, South Korea, 2019.

[75] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, and G. Cucurull, "Llama 2: Open foundation and fine-tuned chat models," *2307.09288*, 2023. [Online]. Available: https://arxiv.org/abs/2307.09288.

[76] F. Galatolo, *vanilla-llama*, 2023. [Online]. Available: https://github.com/galatolofederico/vanilla-llama.

[77] karpathy, *Nanogpt*, [Accessed: 2024-07-11]. [Online]. Available: https://github.com/karpathy/nanoGPT.

[78] A. Gokaslan and V. Cohen, *Openwebtext corpus*, http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[79] P. Bo, *Blinkdl/rwkv-lm: 0.01*, 2021. DOI: 10.5281/zenodo.5196577. [Online]. Available: https://doi.org/10.5281/zenodo.5196577.

[80] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[81] *Roboflow*, 2024. [Online]. Available: https://roboflow.com.

[82] *Gnu affero general public license*, https://www.gnu.org/licenses/agpl-3.0.en.html, [Accessed: 2024-08-12].

[83] K. E. Mitchell, *Copyleft should be scary*, [Accessed: 2024-08-12], 2020. [Online]. Available: https://writing.kemitchell.com/2020/02/02/Scary-Copyleft.

[84] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[85] S. Chen, Z. Lin, and Y. Zhang, "{Selectivetaint}: Efficient data flow tracking with static binary rewriting," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.

[86] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, Nov. 2022.

[87] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2015.

[88] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[89] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, New York, NY, Feb. 2020.

[90] Y. Hu, H. Wang, Y. Zhang, B. Li, and D. Gu, "A semantics-based hybrid approach on binary code similarity comparison," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1241–1258, 2019.

[91] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[92] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li, "{Rapidpatch}: Firmware hotpatching for {real-time} embedded devices," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[93] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "Graphspd: Graph-based security patch detection with enriched code semantics," in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2023.

[94] C. Niesler, S. Surminski, and L. Davi, "Hera: Hotpatching of embedded real-time applications.," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual Conference, Feb. 2021.

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

[96] M. D. Brown, A. Meily, B. Fairservice, A. Sood, J. Dorn, E. Kilmer, and R. Eytchison, "A broad comparative evaluation of software debloating tools," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[97] M. Ali, M. Muzammil, F. Karim, A. Naeem, R. Haroon, M. Haris, H. Nadeem, W. Sabir, F. Shaon, F. Zaffar, *et al.*, "Sok: A tale of reduction, security, and correctness-evaluating program debloating paradigms and their compositions," in *Proceedings of the 28th European Symposium on Research in Computer Security (ESORICS)*, Hague, The Netherlands, Sep. 2023.

[98] J. Christensen, I. M. Anghel, R. Taglang, M. Chiroiu, and R. Sion, "{Decaf}: Automatic, adaptive de-bloating and hardening of {cots} firmware," SEC20.

[99] M. Alhanahnah, R. Jain, V. Rastogi, S. Jha, and T. Reps, "Lightweight, multi-stage, compiler-assisted application specialization," in *Proceedings of the 7th European Symposium on Security and Privacy (EuroS&P)*, Genoa, Italia, Jun. 2022.

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

[101] 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.

TABLE V: Comparison of ZEN's Application to Prior Work.

Recovery Tool[1]	LS [24]	DnD [14]	BTD [15]	AIP [16]	MXR [23]	MXT [23]	ZEN
Model Recovery							
Encrypted Recovery[2]	○	○	○	●	○	◑	●
MR Recovery	◑	●	●	●	●	●	●
Operator Recovery	○	◑	◑	◑	◑	◑	●
PR Recovery	○	○	○	○	○	○	●
Custom Layer Recovery	○	○	○	○	○	○	●
Model Reuse							
Recreates Environment.[3]	○	○	○	○	○	○	●
ONNX Independent	○	○	○	◑	○	○	●
Instrumentable Model	○	◑	◑	◑	○	○	●

● = Full support, ◑ = Partial support, ○ = No support
1: In order from left to right: LibSteal [24], DnD [14], BTD [15], AIP [16], ModelXRay [23] ModelXtractor [23], ZEN (ours)
2: DL system employs encryption to protect model weights, layers, operators, etc.
3: Recreates environment containing all dependencies needed for model reuse is available.

[102] 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.

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

[104] 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.

[105] 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.

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

[107] H. Yoo, S. Kalle, J. Smith, and I. Ahmed, "Overshadow plc to detect remote control-logic injection attacks," in *Proceedings of the 16th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Gothenburg, Sweden, Jun. 2019.

APPENDIX

A. Similarity Score Granularity

Table VI shows the constitutive components making up the mathematical and programmatic similarities for each customized/base model pairs (Columns 1 and 2). Columns 3-5 show the similarity scores between customized models and their attributed models. This directly matches the similarities shown in Figure 3 and Table III.

Columns 6-8 show the architecture (Arch), tensor (TS), and operator (OP) similarities that comprise the overall mathematical representation similarity score. First, we see that the highest similarity is for YoloV5-KD (1.0, 1.0, 1.0 for Arch, TS, OP, respectively). We observe that for each recovered base model relative to its own unified representation in the base model library (i.e., YV5 compared with YV5), the similarity scores for Arch, TS, and OP are exactly 1.0, indicating no difference between the recovered and base model unified representations. We also observe that even if a customized model architecture remains relatively the same (YoloV8-SPD having an Arch similarity of 0.92), the size of tensors/number of weights representing that model can be significantly different as a result of user-defined scaling

TABLE VI: ZEN's Model Attribution Breakdown Showing Mathematical, Programmatic, and Unified Similarity Scores and Their Individual Score Breakdowns.

Base MD[1]	Rec MD[1]	Sim Scores (SS)[2]			Model Attribution[3]				
					MR (SSs)			PR (SSs)	
		MR	PR	UR	Arch	TS	OP	COs	CLS
YV5 [29]	YV5-KD [63]	1.0	1.0	1.0	1.0	1.0	1.0	0.97	1.0
	YV5-HIC [54]	1.0	0.96	0.97	0.84	0.76	0.86	0.98	0.59
	YV5	0.82	0.98	0.94	1.0	1.0	1.0	1.0	1.0
YV7 [65]	YV7-C3 [66]	0.73	0.94	0.89	0.81	0.73	0.59	0.93	0.71
	YV7-GAM [67]	0.68	0.94	0.88	0.77	0.7	0.47	0.93	0.68
	YV7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
YV8 [59]	YV8-GD [68]	0.54	0.86	0.78	0.72	0.73	0.35	0.83	0.39
	YV8-SPD [60]	0.46	0.63	0.59	0.92	0.12	0.42	0.64	0.86
	YV8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
RN [69]	RN-GH [71]	0.63	0.5	0.53	0.81	0.5	0.29	0.5	0.55
	RN-SE [72]	0.47	0.5	0.49	0.71	0.46	0.54	0.5	0.75
	RN	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MNV2 [73]	MNV2-GH [71]	0.32	0.76	0.65	0.33	0.62	0.39	0.72	0.50
	MNV3 [74]	0.51	0.42	0.44	0.66	0.74	0.29	0.33	0.53
	MNV2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.00
nGPT [77]	nGPT-LR [31]	0.55	0.61	0.6	0.94	0.33	0.43	0.52	90.0
	nGPT-RK [79]	0.71	0.76	0.75	0.89	0.84	0.2	0.7	63.6
	nGPT	1.0	1.0	1.0	1.0	1.0	1.0	1.0	100.0
L2 [75]	L2-PT [76]	1.0	0.62	0.72	1.0	1.0	1.0	0.65	62.5
	L3 [21]	0.99	0.62	0.72	1.0	0.95	1.0	0.63	87.5
	L2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	100.0

1: The base model from which the recovered models may be derived from.
2: Overall similarity scores (MR, PR, UR) between base and recovered models.
3: Model attribution similarity scores (%) for the MR (arc, tensors, operators) and PR (code objects, modules) for the recovered model to the base model.

changes (YoloV8-SPD has a TS similarity of 0.12). For model operators we observe that the mean similarity is 0.55 with a standard deviation of 0.28, indicating that operator similarity, even for models within the same family, may vary significantly (up to 28%). This indicates that the mathematical representation similarity should **not** be taken in isolation for model attribution.

Columns 9-10 show the code object similarity (COs) and class similarity (CLS) scores for ZEN's model attribution. We observe that across all customized models, the average code object and class similarities were 0.58 and 0.7, respectively. This indicates that practitioner-refined DL systems only modify/add around 42% of the functions and 30% of the classes relative to the base implementation. This further highlights that while mathematical representation similarity can vary significantly, programmatic representation similarity directly complements the mathematical representation similarity, ultimately enabling accurate attribution.

B. ZEN Evasion

ZEN relies on unified model representation recovery for model attribution. An adversary can attempt to obfuscate a DL system such that unified model representation recovery (and consequently attribution) is impeded. Fortunately, even if an adversary deployed a DL system containing obfuscated data structures, ZEN was built to target DL systems at the interpreter level, meaning that obfuscation of the framework/DL model code would not affect ZEN's recovery. ZEN recovers byte-code level implementations and handles recovery of all object/function definitions at a lower level than can be controlled by all but the most capable adversary. In other words, framework obfuscation does not affect ZEN's recovery, as commodity frameworks do not make changes to the interpreter for any aspect of model implementation.Any attempt to thwart ZEN in this way would entail an adversary making widespread changes to data structures at the interpreter/compiler level. We consider this to be out of the scope of this work, as making such an attempt would put a large burden on the adversary. An adversary can attempt to thwart ZEN by introducing bloat into the DL system (e.g., other models, unused code, etc.) to try to cause ZEN to misattribute the model.

1) Evasion Via Malicious DL System Bloating: Here, we aim to highlight why adversaries attempting to evade ZEN via intentional DL system bloating will fail. First, ZEN analyzes DL systems from *memory*, meaning that for interpreter-based DL systems, unless the code is actively used by the adversary at runtime, it will not be tracked by the GC and therefore will be avoided during mathematical and programmatic representation recovery. For adversaries attempting this on non-interpreter-based DL systems, a variety of software debloating methodologies and tools [96]–[99] can be used in tandem with ZEN to eliminate the bloat introduced by the adversary.

Even if the adversary successfully introduced bloat into the system and loaded it into memory, ZEN uses a mathematical representation-guided recovery approach for programmatic representation recovery. ZEN first finds the model object and subsequently finds the model layers, layer types, weights, etc. Then, by identifying code referencing mathematical representation-related data structures, ZEN constructs the programmatic representation of the DL system. This implies that in order for the adversary to evade ZEN's programmatic representation recovery (or misguide it) they would also have to load an entire model object into the DL system at runtime. This is not only cost ineffective (increasing latency of the DL system on serverless deployments) but also unrealistic, as large models already monopolize a DL system's RAM/VRAM. An adversary would not only have to load multiple model objects into memory but would also need to fall back on interpreter-level obfuscation techniques to hide the model, as ZEN would recover the programmatic representations for *all* model objects in memory.

C. Memory Acquisition

As described in prior work [16], forensic investigators employ a variety of tools and techniques to acquire memory images from diverse systems. For Linux environments, popular tools include Microsoft's AVML [50] and LiME [48]. Memory acquisition from Android devices can also be performed using LiME, enabling analysis of deployed mobile models with ZEN. Specialized forensic techniques exist for automotive systems [100]–[104], while embedded platforms like Arduino and Raspberry Pi often possess built-in capabilities for dumping volatile memory. Furthermore, tools have been developed to acquire and analyze memory from Programmable Logic Controllers (PLCs) within Industrial Control Systems (ICS) for security purposes [105]–[107]. In the experiments conducted for this paper, CPU memory was

collected from the running DL system at inference time for input into ZEN. A complete CPU volatile memory image was acquired using LiME [48], reflecting the model's state when the system is running.

D. Forensic Evidence

After all differences between the recovered model and base models are identified at the lowest level (changes in byte-code, attributes, etc.), the investigator must be made aware of the patches made to the base model. This provides essential forensic evidence (e.g., for legal proceedings), as exemplified in §V-D. ZEN can therefore be used to present to the investigator, in a human-readable format, the source code changes for all differences between the base and deployed DL systems. ZEN can also directly output source code similarities, highlighting how much code was reused in the unknown DL model (such as for the purposes of copyright infringement). Referring back to Algorithm 2, for each pair in Ω, ZEN outputs to the investigator the difference recorded in each member of ϵ. The decompiled byte-code difference and changed attributes are presented in a high-level format (i.e., changed vars: foo, bar, localvars: foo1, bar1...). Likewise, each member in Υ is presented to the investigator alongside its byte-code and attributes.

E. Ethics Consideration

We directly recovered and attributed models from large open-source AI marketplaces (e.g., Hugging Face [2], Github [3], etc.) and did not target proprietary models for which we do not have access. We *do not* claim that we engineered the models ourselves. Models were retrieved from SOTA open-source work and were issued primarily with MIT licenses. In our work we explicitly cite all models investigated by ZEN and give credit to the authors. We *do not* use private or unauthorized models for ZEN's base model library building or evaluation.