

A Progressive Transformer for Unifying Binary Code Embedding and Knowledge Transfer

Hanxiao Lu¹, Hongyu Cai², Yiming Liang², Antonio Bianchi², Z. Berkay Celik²

¹Columbia University, hl3424@columbia.edu

²Purdue University, {hongyu, liang328, antoniob, zcelik}@purdue.edu

Abstract—Language models have recently been applied to binary analysis tasks, such as function similarity detection and function signature recovery. These models typically employ a two-stage training process: pre-training via Masked Language Modeling (MLM) on machine code and fine-tuning for specific tasks. While MLM helps to understand binary code structures, it ignores essential code characteristics, including control and data flow, which negatively affect model generalization. Recent work leverages domain-specific features (e.g., control flow graphs and dynamic execution traces) in transformer-based approaches to improve binary code semantic understanding. This approach, however, involves complex feature engineering, a cumbersome and time-consuming process that can introduce predictive uncertainty when dealing with stripped or obfuscated code, which leads to a performance drop. We introduce PROTST, a novel transformer-based methodology for binary code embedding. PROTST employs a hierarchical training process based on a unique tree-like structure, where knowledge progressively flows from fundamental tasks at the root to more specialized tasks at the leaves. This progressive teacher-student paradigm allows the model to build upon previously learned knowledge, resulting in high-quality embeddings that can be effectively leveraged for diverse downstream binary analysis tasks. The effectiveness of PROTST is evaluated in seven binary analysis tasks, demonstrating an average of 14.8% improvement in F1 and MRR compared to traditional two-stage training, and a 16.6% improvement when analyzing obfuscated code.

Index Terms—Binary Analysis, Large Language Models, Transfer Learning.

I. INTRODUCTION

Deep learning, particularly NLP-inspired models, such as RoBERTa [1] and GPT-based transformers [2], have significantly affected diverse downstream binary analysis tasks, e.g., function similarity detection, indirect call recognition, and function signature recovery. These models leverage transformer architectures, which effectively capture contextual information and long-range dependencies in sequential binaries. The adoption of such models in learning-based binary analysis has opened new avenues to improve the accuracy and efficiency of these tasks [3], [4], [5], [6].

These models typically follow a two-stage training process: (1) an initial pre-training phase via Masked Language Modeling (MLM) on machine code and (2) a subsequent fine-tuning phase for a specific task. In the pre-training phase, the model learns general representations of the code by predicting masked tokens, which helps to understand the structure and patterns within binary code. The fine-tuning phase, on the other hand, involves training the model on a labeled dataset specific to the target task,

allowing it to adapt its learned representations to that particular downstream task. XDA [3] and Binprov [7] use this strategy for disassembly and compiler provenance tasks. However, in this context, MLM only analyzes the sequential order and relationships between tokens, neglecting binary code's inherent structure and knowledge, such as control and data flow [8], [9]. Recent studies [5], [10], [8], [4] showed that, without a mechanism to incorporate such structure and knowledge, the accuracy of target tasks may drop.

To address this issue, recent transformer-based binary analysis frameworks [10], [8], [9], [5], [6], [11] incorporate a wide range of domain-specific knowledge into the traditional two-stage architecture by concatenating different high-level modalities as input, e.g., assembly language, control flow graphs (CFG), data flow graphs (DFG), and dynamic execution traces. By integrating these additional features, the model gains a richer understanding of binary code semantics; thus, it may improve the performance of downstream tasks.

Extracting these features, however, necessitates sophisticated reverse engineering tools and specialized knowledge for each CPU architecture. This complexity makes them cumbersome to use and limits their effectiveness in diverse binary datasets. Furthermore, popular reverse engineering tools struggle with stripped or obfuscated code, and misidentify function boundaries and assembly instructions [12]. These errors introduce noise and inaccuracies into the extracted features, ultimately degrading the data quality fed to deep learning models. Moreover, some approaches [9] indiscriminately incorporate a wide array of high-level features, e.g., operand type, operand read/write status, and FLAGS register status during pre-training, regardless of the target tasks the model will be fine-tuned for later. This indiscriminate incorporation can include information not relevant to the specific downstream task.

These limitations raise a critical research question: *Can we effectively capture the knowledge inherent in binary code without relying on complex reverse engineering and feature engineering, and instead, introduce an appropriate amount of task-specific knowledge, avoiding the disadvantages of including possible modalities into the model during pre-training?* To answer this question, in this paper, we introduce PROTST, a Progressive Teacher-Student Binary Analysis framework that transfers knowledge between binary analysis tasks to develop high-quality embeddings. We observe that inherent binary knowledge does not necessarily need to be learned from high-level modalities but can be effectively captured through a

step-by-step progression of causally related binary tasks.

To achieve this, PROTST adopts a hierarchical tree structure, where knowledge progressively flows from fundamental tasks at the root to more specialized tasks at the leaves. That is, lower-level tasks, such as instruction and function boundary recovery, reside near the root, while more specialized tasks, such as function similarity detection and function name prediction, are placed toward the leaves. Each node in this structure functions as a student relative to its predecessor node. The model acquires foundational knowledge from its teacher node and enhances it with task-specific insights before teaching its downstream node, which addresses a more challenging binary task. For example, the function boundary recovery task acts as a student relative to the instruction boundary recovery task. Once it learns from its teacher, it becomes the teacher of the function signature prediction task, which enables the model to leverage the extracted function boundaries for more accurate predictions. The knowledge of the function signature then becomes a teacher for more advanced tasks, such as function similarity detection and function name prediction.

This hierarchical design ensures a natural and logical progression of knowledge, following the teacher-student learning paradigm [13], [14]. By placing related tasks with direct logical connections close to each other, the model benefits from a coherent flow of information. Unlike traditional two-phase training architectures [5], [9], [5], [6], [11], where MLM pre-training is the only strategy irrespective of the downstream task, our progressive teacher-student framework systematically guides the model from basic to advanced tasks. This step-by-step progression, facilitated by the continuous transfer of knowledge through model weight refinement, generates semantically rich embeddings that are fine-tuned for a variety of binary analysis tasks. To our best knowledge, PROTST is the first work to build knowledge transfer between binary analysis tasks within learning-based binary code embedding.

PROTST is evaluated across seven binary analysis tasks with diverse datasets and compared its performance against state-of-the-art methods under identical experimental settings. The evaluation demonstrates significant advantages of the teacher-student paradigm over a traditional two-stage training architecture. When the teacher-student approach is introduced, we observe an average of 14.8% with at least 5% improvement in validation score (F1 and MRR) in all tasks. Additionally, PROTST achieves a 16.6% performance improvement when analyzing obfuscated binaries.

In summary, we make the following contributions.

- We introduce a progressive teacher-student paradigm for binary analysis tasks. This paradigm facilitates efficient knowledge transfer from fundamental tasks to more complex ones, enabling the model to progressively build a hierarchical understanding of binary code.
- To realize this paradigm, we introduce a model architecture that consists of three components: an embedding module that transforms binary code into a high-dimensional representation, a transformer backbone model that cap-

tures raw bytes relationships, and task-specific heads for different objectives.

- We evaluate PROTST on seven diverse binary analysis tasks, and show that it improves validation scores by an average of 14.8% over traditional two-stage training and achieves a 16.6% performance gain when analyzing obfuscated binaries.

PROTST is publicly available at [15] for use and validation.

II. BACKGROUND AND RELATED WORK

Binary Code Embedding (BCE) is a technique that maps raw binary code into lower-dimensional vectors, allowing these embeddings to be used for various binary analysis tasks. Below, we present BCE approaches by grouping them into learning-based and transformer-based.

Learning-based BCE. With the increasing availability of large datasets and advances in deep learning, most BCE approaches initially used learning-based approaches. These methods can be broadly categorized into three groups: (1) Sequence-based, (2) CNN-based, and (3) GNN-based. Sequence-based models such as SAFE [16], EKLAVYA [17], and Bi-RNN [18] use RNNs (e.g., LSTMs [19], GRUs [20]) for tasks such as code similarity detection, function signature prediction, and boundary detection. Malconv2 [21] and o-glasses [22] use 1-d CNNs to capture binary code embeddings for tasks such as malware classification and compiler provenance. Additionally, IMCFN [23] takes a different approach, using 2-dimensional CNNs to embed binaries as images specifically for malware classification. Techniques such as Gemini [24], DeepDi [25], and Structure2Vec [26] use GNNs to model binary code using graph representations, e.g., CFGs, DFGs.

Transformer-based BCE. Recent advancements in NLP, particularly transformer models, have sparked a series of BCE techniques. These methods leverage the transformer's self-attention mechanism to capture long-range dependencies and complex patterns within raw binary code. They typically involve a two-stage training process: pre-training with MLM followed by fine-tuning on the specific target task. Models such as XDA [3] and BinProv [7] directly apply transformers to raw byte code for tasks, e.g., disassembly and compiler provenance analysis. Other models recognize the importance of including the inherent knowledge of binary code to improve understanding. They achieve this by working on assembly code and embedding information from CFGs and DFGs into the model. This approach is used by Palmtree [10] and jTrans [8] to address tasks such as function signature recovery and function similarity detection. Other methods, including Trex [5], BinBert [27], kTrans [9] and Symmlm [4], incorporate dynamic code behavior, register/operand details, or caller/callee information to improve function similarity detection and function name prediction.

III. TEACHER-STUDENT LEARNING FOR BINARY TASKS

We introduce PROTST, a novel transformer-based framework for binary analysis tasks, which leverages a teacher-student learning paradigm [13], [14]. PROTST organizes a series of

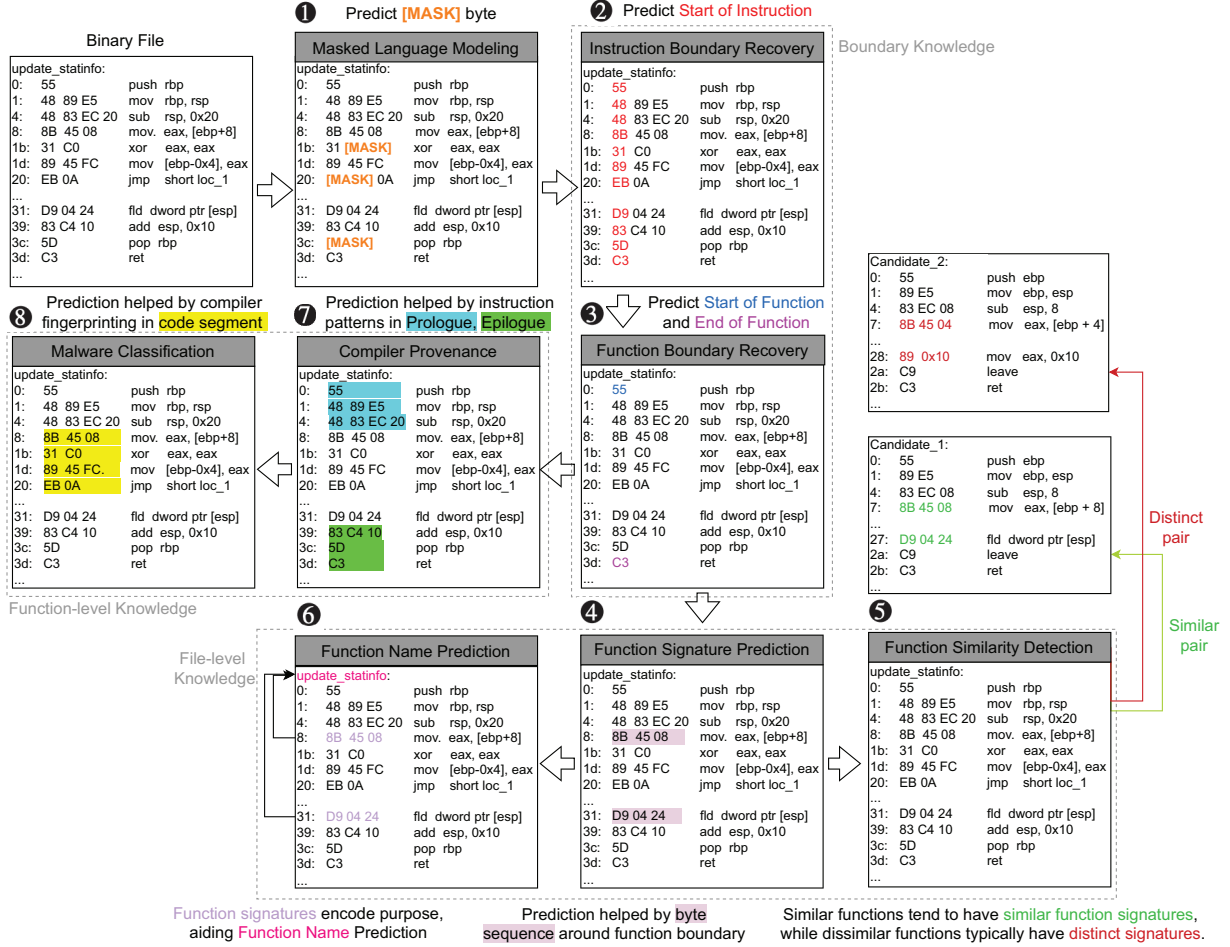


Fig. 1: The progressive *teacher-student* learning of ProTST. Tasks are hierarchically structured to leverage foundational knowledge for more complex tasks. Each task employs a transformer, with model weights serving as interfaces between adjacent nodes. The system operates solely on the raw byte sequence (address and assembly are shown for illustration only).

tasks in progressive order, where each subsequent task builds on the knowledge gained from its predecessors.

Here, each task specializes in its domain, acting as a focused teacher on the subsequent student task. This progressive knowledge transfer provides a deeper understanding of binary code than cramming various high-level modalities (e.g., CFG, execution trace, register information) into a single model. In addition, the collaborative nature of ProTST allows each stage to build upon the knowledge acquired by previous stages. This results in an effective transfer of knowledge regarding semantic patterns among tasks, eliminating the need for customized embeddings, complex features and reverse engineering, or explicit knowledge of the binary code structure.

Motivation and Approach Overview. Figure 1 shows the architecture of ProTST, which conceptualizes the relationships between binary analysis tasks as a series of teacher-student paradigms. In the following, we will detail how the knowledge acquired and representations learned during the execution of

one task (the teacher) can be transferred and leveraged to enhance the performance on a subsequent task (the student). We will empirically validate these relationships in Section V.

ProTST is composed of two main stages: (a) a Masked Language Modeling (MLM) stage (1) to understand the syntax of the language at the byte level, and (b) a novel Binary Knowledge Accumulation (BKA) stage (2-8) to capture and transfer inherent knowledge within binary code across various tasks. In the BKA stage, the model learns three types of knowledge: boundary knowledge (2-3), function-level knowledge (4-6), and file-level knowledge (7-8).

The initial MLM stage (1) acts as the first teacher. It trains a base model to predict masked bytes within the binary, equipping the model with a fundamental understanding of the binary's structure and content, akin to learning the alphabet of a new language. The student model then addresses the instruction boundary recovery (2), leveraging the foundational knowledge from the MLM stage to identify boundaries between individual instructions. This crucial step enables the model to group bytes

into meaningful instruction sequences, similar to parsing words from sentences. Building on this foundation, function boundary recovery (3) becomes the next teacher, enhancing the student's ability to delineate functional blocks within the binary based on the previously learned instruction boundaries. Mastering function boundaries is essential for further analysis reliant on the program's functional organization, akin to understanding paragraphs and their purposes within a text.

Once the student model has grasped these two levels of boundary knowledge, it delves deeper into function-level knowledge. This includes function signature prediction (4), which acts as a high-level summary of a function's purpose, similar to titles for chapters in a book. Accurate predictions offer valuable insights into a function's role within the program. Precise boundary knowledge is crucial for accurate signature prediction. Recognizing where functions start and end helps the model isolate and focus on the relevant bytes that make up a function. This focus ensures that the model analyzes only the pertinent parts of the code, improving its understanding of the function's structure and behavior.

In addition, identifying individual instructions within a function offers a more granular view of its operations. This perspective helps the model determine the number of arguments a function accepts and the data types it returns. In essence, both function and instruction boundary knowledge provide a strong foundation for accurate function signature prediction.

The extracted semantics from a function's signature form a unique fingerprint. This fingerprint captures a high-level overview of the function's purpose and its interactions within the program. By analyzing and comparing these fingerprints, the model can detect functions with similar functionalities, even if their internal implementations differ. This capability is important for function similarity detection (5). Leveraging these fingerprints facilitates the model's discovery of shared functionalities and code reuse patterns across functions. Furthermore, the patterns derived from function signatures offer valuable clues about a function's purpose. This information is beneficial for the task of function name prediction (6). By recognizing similar signatures, the model can infer the likely role and behavior of a function, guiding it to assign descriptive, human-readable names that reflect each function's purpose.

Beyond understanding the internal structure and functionality of binaries, PROTST also extends its focus to file-level knowledge, with particular attention to compiler provenance (7) and malware classification (8). Here, the two levels of boundary knowledge acquired during pre-training come into play. The boundary knowledge facilitates the model in detecting distinct patterns and traits associated with specific compilers. Function prologues and epilogues often contain unique sequences of instructions that are characteristic of the compiler used. These, along with other internal instruction patterns, provide valuable clues that help to pinpoint the likely compiler that generated the binary. Understanding compiler-specific traits reveals optimizations and behaviors that influence the binary's performance and structure. In the realm of security, particularly in malware classification, this knowledge

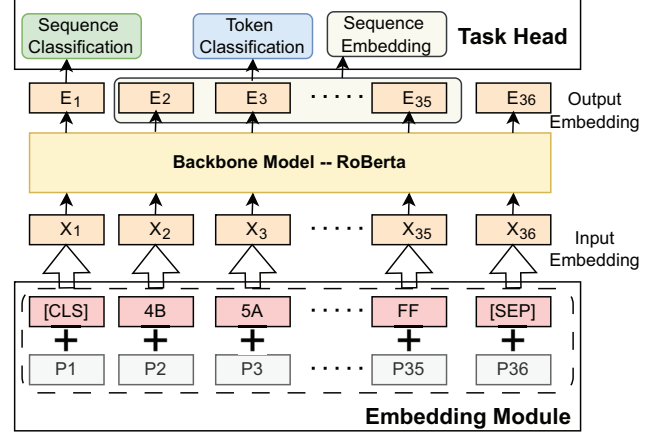


Fig. 2: Model architecture of PROTST.

is crucial. The design of PROTST assumes that malware from the same family is often compiled using the same (or a similar) compiler [28]. Consequently, identifying the compiler used can be instrumental in tracing malware's origins and family.

A. Model Architecture

Figure 2 illustrates the model architecture of PROTST across all binary analysis tasks. This unified architecture enables the transfer of knowledge and representations between tasks. The model comprises three components: (1) embedding module, (2) backbone model, and (3) task head module.

1) *Embedding Module*: The embedding module is the first step for processing binary code. It transforms the raw data of a binary file, a sequence of bytes, for further analysis by the model. We define the input x as a sequence of byte tokens of size n : $x = \{0x00, \dots, 0xff\}^n$. Each input byte $x_i \in x$ is represented as a one-hot encoded vector, e.g., $a3$ is encoded as a 256 dimensional vector with all 0s but single 1 at position 163.

Beyond byte-level information, capturing relative positions within the code is crucial to understanding its meaning. Unlike natural language, where the swapping of two words can roughly preserve the same semantic meaning, swapping two bytes can significantly change the instructions. To address this issue, a widely used learned positional encoding method [1] is employed. This method first transforms the one-hot encoded byte token x_i through an embedding $E_{byte}(x_i)$. This embedding captures the semantic meaning of the individual byte. We then incorporate positional information into the model by applying a learned positional encoding $E_{pos}(i)$ based on the specific position i of the byte token x_i . Lastly, a final input embedding is created, $E_i(x_i)$, for the byte token by combining:

$$E_i(x_i) = E_{byte}(x_i) + E_{pos}(i) \quad (1)$$

2) *Backbone Model*: The backbone model is the core component for processing the embedded binary code generated by the embedding module. We adopt a multi-layer transformer encoder, RoBERTa [1], as the backbone model. The transformer is a bidirectional language model based on the self-attention

mechanism, which allows it to capture contextual dependencies between byte embeddings at different levels of abstraction.

The self-attention mechanism computes a weighted sum of the input embeddings, where the pairwise similarity between the tokens determines the weight. Let $X = [x_1, x_2, \dots, x_n]$ be a sequence of byte embeddings, where $x_i \in \mathbb{R}^d$ and d is the embedding dimension. The self-attention mechanism can be expressed as follows:

$$\text{Attention}(X) = \text{softmax} \left(\frac{XW_q(XW_k)^T}{\sqrt{d_k}} \right) XW_v \quad (2)$$

where W_q , W_k , and W_v are learned weight matrices, and d_k is a scaling factor.

In the context of binary analysis, each x_i represents an embedded byte. The self-attention mechanism allows the model to weigh the importance of each byte in the context of all other bytes in the sequence. This enables the model to learn complex relationships between bytes, even if they are not directly adjacent in the sequence. For instance, the model can learn that a byte representing a function call is related to the bytes representing the corresponding function definition, regardless of their distance in the binary.

Within our framework, we define a sequence of T binary analysis tasks, with each task $t \in \{1, 2, \dots, T\}$ employing a dedicated instance of the RoBERTa transformer, denoted as M_t . The knowledge transfer process proceeds sequentially through this task chain. More specifically, let Θ_t denote the set of parameters for the model M_t . For the initial task ($t = 1$), the model is trained on a labeled dataset D_1 , optimizing its parameters to minimize a task-specific loss function L_1 :

$$\Theta_1^* = \underset{\Theta_1}{\operatorname{argmin}} L_1(M_1(\Theta_1), D_1) \quad (3)$$

For subsequent tasks ($t > 1$), the knowledge acquired by the previous model M_{t-1} is transferred to initialize the parameters of the current model M_t :

$$\Theta_t^{\text{init}} = \Theta_{t-1}^* \quad (4)$$

Following this initialization, further fine-tuning is performed on the task-specific dataset D_t , optimizing the parameters to minimize the loss L_t :

$$\Theta_t^* = \underset{\Theta_t}{\operatorname{argmin}} L_t(M_t(\Theta_t), D_t) \quad (5)$$

This iterative process of knowledge transfer and fine-tuning allows each model M_t to benefit from the knowledge accumulated by its predecessors in the task chain, leading to a progressive refinement of representations and improved performance on downstream tasks.

In contrast to the common practice of freezing certain model parameters during fine-tuning, we allow all parameters in the backbone to be updated during training for each task t [29]. This complete fine-tuning approach provides the model with greater flexibility to adapt to the specific nuances of each task and leads to improved performance (See Section V).

3) *Task Head Module*: The BKA stage of ProTST encompasses a diverse set of tasks to accumulate knowledge about binary code. Each task addresses a specific aspect of binary analysis, and to facilitate this learning process, we equip the backbone model with distinct task-specific “heads”. The heads serve as interfaces that enable the model to apply the knowledge from the backbone to the specific requirements of each task.

ProTST addresses seven distinct binary analysis tasks within the BKA stage. We employ three types of task-specific heads, each tailored to the particular classification problem.

Token-Level Classifier. With this head, we focus on fine-grained analysis of the byte sequence, where the goal is to assign a class label to each individual byte (token). Tasks including instruction and function boundary recovery (tasks ②-③) fall into this category. For each token x_i in the input sequence x , the token-level classifier processes its corresponding hidden state h_i from the backbone model and produces an output y_i .

Sequence-Level Classifier. We use this head for tasks that necessitate the characterization of an entire byte sequence, such as predicting function names, identifying compilers, or classifying malware families (tasks ④, ⑥-⑧). It operates on the final hidden state $h_{[CLS]}$ corresponding to the [CLS] token produced by the backbone model.

Sequence Embedding Head. We use this head for the task of function similarity detection (⑤). It generates a fixed-length embedding representation e for an entire byte sequence x by aggregating (e.g., averaging) the output embeddings $h_1, h_2, \dots, h_n$ from the backbone model. These sequence embeddings can then be compared using cosine similarity or other suitable distance metrics to assess the functional similarity between different code segments.

Although ProTST involves a multi-stage pre-training process, it utilizes only one instance of RoBERTa during inference. This design allows for efficient adaptation to new tasks. If a new binary analysis task needs to be added to the hierarchy, only the new task after the insertion point requires further training. For instance, if a new task needs to be placed after function boundary recovery, we can leverage the pre-trained checkpoints stored at that stage and continue fine-tuning the new task.

IV. IMPLEMENTATION AND EVALUATION SETUP

We evaluate ProTST using seven binary tasks, each performed on a distinct dataset. Since binaries within a single dataset may exhibit similar patterns, this diversity in data patterns is crucial to demonstrate ProTST’s generality. In this paper, we focus on the x86 architecture, as it is prevalent in BCE research involving multiple tasks [8], [9], [10]. The summary of each dataset is provided in Table I.

A. Model Configuration

We configure ProTST with the pre-trained RoBERTa model, using its default settings of 12 layers, 12 attention heads per layer, and a hidden dimension of 768. The maximum input sequence length is set to 512 tokens.

Pre-training. In this stage, we employ either a binary or project-level split depending on the task. For instruction/function

TABLE I: Binary datasets used to evaluate PROTST.

Dataset	Data Size	Task
Binutils [30]	1 project	Masked Language Modeling
SPEC CPU [31], [32]	2.01G bytes	Inst./Func. Boundary Prediction
BAP [12]	345M bytes	Func. Boundary Prediction
BinKit [33]	75M functions	Compiler Prov./Func. Signature Prediction
SymIml [4]	1.44M functions	Func. Name Prediction
Binarycorp-3M [8]	404K functions	Func. Similarity Detection
BIG2015 [34]	10860 files	Malware Classification

boundary recovery and malware classification, a binary-level split is used to ensure that the model is evaluated on entirely unseen binaries. For other tasks, a project-level split is used to ensure that data from different projects are kept separate during pre-training and evaluation.

In both cases, the split ratio is 90% for pre-training and 10% for evaluation. This setup aims to assess the model’s ability to generalize to unseen data and avoid overfitting. During pre-training, each task (teacher) is trained with a batch size of 96 samples for 20 epochs. The MLM pre-training task adopts a replacement ratio of 0.5, which aligns with XDA [3].

Fine-tuning. For fine-tuning, we randomly select 100K samples from the evaluation dataset of each task, except for malware classification, where we use 10K samples due to limited data availability in BIG2015. The selected samples are then split into fine-tuning and testing sets. For tasks involving instruction boundaries, compiler provenance, function signatures, and function similarity detection, we allocate 1% of the samples for fine-tuning and 99% for testing. With this train-test ratio, we aim to minimize overfitting, better generalize to unseen data, and expose the pre-trained teacher models to a much larger and more diverse dataset than the student models to enable effective knowledge transfer. A less strict split of 10% for fine-tuning and 90% for testing is employed for other tasks. Similar to pre-training, fine-tuning involves processing the data in batches of 96 samples for 100 epochs.

B. Evaluation Metrics

To assess PROTST’s performance in binary analysis tasks, we employ established evaluation metrics from previous work.

Following recent work [3], [4], [6], the primary metric used to evaluate most of the tasks in PROTST is the F1 score. To account for potential class imbalances that might skew performance assessments, we employ the macro-F1 score variant. This metric is computed by first calculating the F1-score for each individual class, which is the harmonic mean of precision and recall.

For the task of function similarity detection, we adopt the Mean Reciprocal Rank (MRR) from a recent work [8]. MRR quantifies how well the model ranks the most similar function (ground truth) relative to others for a given query function.

We note that some tasks within our framework have distinct subtasks. Function signature prediction, for example, includes predicting both argument count and return type. Similarly, function similarity detection involves evaluating similarity at varying sizes (32 and 10K functions). To fully assess the PROTST’s performance, we report metrics for each subtask.

V. EVALUATION

We evaluate PROTST in seven diverse binary analysis tasks using the datasets and evaluation metrics described in Section IV. We present our findings through several key research questions:

- RQ1** Does the progressive teacher-student paradigm employed by PROTST yield improved performance on downstream binary analysis tasks compared to established baselines?
- RQ2** To what extent does PROTST generalize to code obfuscation techniques?
- RQ3** How does the order of tasks in the BKA stage influence knowledge transfer and impact overall performance?

To ensure a fair comparison, in the experiments, all models undergo the same fine-tuning procedure as for PROTST, detailed in Section IV-A. All experiments were performed on a dedicated server with eight AMD EPYC 7543 32-core processors, one A100 GPU, 32GB memory, and 1TB SSD.

A. RQ1: Overall Binary Task Effectiveness

We present the performance of PROTST in each individual task and compare it with the state-of-the-art method of each task. For each task, we report the performance metrics of the models for each class within the dataset.

To further investigate the efficacy of the teacher-student paradigm, we introduce a variant model, PROTST₀. In contrast to PROTST, which leverages fine-tuning on student tasks to refine the transferred knowledge, PROTST₀ directly leverages the embeddings generated by the pre-trained teacher model. This approach aligns with the principles of zero-shot learning [35], in which a model is evaluated on unseen tasks without any task-specific adaptation. The performance of PROTST₀ serves as a critical indicator of knowledge transferability by the teacher model, as it relies solely on the teacher’s knowledge to generalize to novel tasks.

1) Instruction Boundary Recovery: The instruction boundary recovery task aims to classify each byte within a binary file as either “Start” (the first byte of an assembly instruction) or “Middle” (all other bytes within an instruction). We evaluate the performance of PROTST against two baselines: the Bidirectional Recurrent Neural Network (Bi-RNN) based method [18] and DeepDi [25], a graph neural network (GNN)-based disassembler. It is important to note that DeepDi is a commercial tool, and we leverage its functionality through its provided API without fine-tuning the model under the same experimental setup used for the other methods.

Table II summarizes the results for this task. Our analysis yields several key observations. First, PROTST achieves the highest average F1-score of 88.4%, outperforming both Bi-RNN (66.4%) and DeepDi (88.1%). Although DeepDi has been trained on a larger dataset of binaries compared to the fine-tuning setting used for PROTST, it still exhibits lower performance in identifying Start classes, which represent a minor class within the dataset (DeepDi: 76.5% vs. PROTST: 82.7%). This improvement is due to the knowledge transferred from the MLM pre-training stage to instruction boundary

TABLE II: F1-scores on instruction boundary recovery.

Model	Class		Average
	Start	Middle	
Bi-RNN [18]	0.443	0.885	0.664
DeepDi [25]	0.765	0.998	0.881
PROTST ₀	0.61	0.906	0.758
PROTST	0.827	0.941	0.884

recovery. Second, PROTST₀ exhibits competitive performance over the Bi-RNN baseline; it yields an average F1-score of 75.8%. This shows that the RoBERTa-based architecture captures local patterns within binary code, even without task-specific fine-tuning. Lastly, a consistent trend across all models is the performance in classifying Middle bytes compared to Start bytes. This disparity is due to the inherent class imbalance, where middle bytes are significantly more prevalent than start bytes, which poses a challenge for models to learn discriminative features for the less frequent class.

2) *Function Boundary Recovery*: The function boundary recovery task aims to classify each byte within a binary file as either “Start of Function”, “Middle of Function”, or “End of Function”. We evaluate PROTST against the same methods used in instruction boundary recovery: Bi-RNN [18] and DeepDi [25]. It is important to acknowledge a limitation in DeepDi’s API for this task. It cannot differentiate between Start and End classes. Consequently, we will report the average F1-score for the combined Start/End class for DeepDi.

Similar to the instruction boundary recovery task, as shown in Table III, all models exhibit near-perfect performance on the dominant Middle class but encounter difficulties in classifying the less frequent Start and End classes due to the inherent class imbalance in the dataset. Yet, PROTST significantly outperforms both DeepDi and Bi-RNN, achieving an average 91.5% F1 score. Although DeepDi has been trained on a larger data set of binaries compared to the fine-tuning setting used for PROTST, it still exhibits lower performance in identifying the Start and End classes (DeepDi: 74.1% vs. PROTST: 84.9% at Start and 89.7% at End). This improvement is attributed to the additional instruction boundary knowledge transferred from the previous stage, which provides valuable contextual information for function boundary recovery. Moreover, PROTST₀ shows a rudimentary ability to classify function boundaries and achieves F1-scores of 6.6% and 4.0% for the Start and End classes. This surpasses the Bi-RNN baseline, which fails to identify any patterns in these classes; this shows the effectiveness of the RoBERTa-based architecture in capturing function boundaries even without task-specific fine-tuning.

3) *Compiler Provenance*: The compiler provenance task involves identifying both the compiler (GCC or Clang) and its specific optimization level (O0, O1, . . . , Ofast) used to generate a given binary file. We evaluate the performance of PROTST against two established methods: O-glasses [22], which employs 1D convolutions to analyze raw binary sequences for provenance, and Binprov [7], which uses a transformer-based architecture with a traditional two-stage training approach that operates directly on raw bytes.

TABLE III: F1-scores on function boundary recovery.

Model	Middle	Class		Average
		Start	End	
Bi-RNN [18]	0.997	0	0	0.332
DeepDi [25]	0.999	0.741	0.741	0.827
PROTST ₀	0.997	0.066	0.04	0.368
PROTST	0.999	0.849	0.897	0.915

Table IV presents the results for this task. PROTST achieves an average F1-score of 46%, outperforming both Binprov (41.6%) and O-glasses (12.9%). Performance improvement is evident in all classes. To better understand the impact of knowledge transfer within PROTST, we compare its results with PROTST₀, which, despite outperforming O-glasses with an average F1-score of 15.3%, exhibits a clear weakness in identifying certain compiler-optimization combinations—particularly GCC-O3, GCC-Os, GCC-Ofast, and Clang-Os—where it achieves a zero F1-score. This suggests that fine-tuning is crucial for achieving better generalization. Our results also demonstrate that classifying binaries compiled with Clang is more challenging than those compiled with GCC, as all models exhibit lower performance on Clang-compiled binaries.

4) *Malware Classification*: In this task, we evaluate PROTST in a malware classification task to identify the specific malware class within a given binary file. We compare PROTST against two established baselines: IMCFN [23], which uses a 2D-CNN to extract image-like features from byte sequences, and Malconv2 [21], which employs a 1D-CNN to directly extract features from the raw 1D representation of the malware bytes.

As shown in Table V, PROTST outperforms both IMCFN (59.4%) and Malconv2 (64.6%), achieving an average F1-score of 80.1%. We observe that performance improvement is due to knowledge transfer enabled by the teacher-student paradigm and the powerful representation learning capabilities of the RoBERTa backbone, which together lead to learning discriminative representations for each class. However, the relatively small size (10K samples) of the BIG2015 dataset limits the ability of PROTST₀ to fully leverage the knowledge transferred from the teacher model without fine-tuning, leading to lower performance than the other models. Notably, all models struggle with the Simda class, as Simda samples comprise only 0.4% of the total dataset.

5) *Function Signature Prediction*: The function signature prediction task involves predicting two crucial aspects of a function’s signature: (a) the return type and (b) the number of arguments it accepts. We evaluate PROTST against two established methods: EKLAVYA [17], which uses RNNs to learn function type signatures, and Palmtree [10], which leverages a transformer-based architecture that incorporates data/control flow information on assembly code.

Table VI summarizes the results. PROTST outperforms both EKLAVYA and Palmtree in both number of arguments prediction, 76.3% vs. 34.9% and 34.0%, and return type prediction, 62.6% vs. 22.6% and 35.2%. While Palmtree reports high accuracy in its experimental setting (237 binaries for training and 14 binaries for testing), it demonstrates less adaptability to

TABLE IV: F1-scores on compiler provenance.

Model	GCC						Clang						Average
	O0	O1	O2	O3	Os	Ofast	O0	O1	O2	O3	Os	Ofast	
o-glasses [22]	0.157	0.346	0.04	0.005	0.013	0.079	0.397	0.137	0.001	0	0.221	0.157	0.129
Binprov [7]	0.888	0.663	0.27	0.221	0.517	0.228	0.918	0.321	0.228	0.125	0.389	0.231	0.416
PROTST ₀	0.485	0.317	0.048	0	0	0	0.578	0.243	0.043	0.1	0	0.016	0.153
PROTST	0.902	0.713	0.293	0.327	0.587	0.267	0.938	0.425	0.183	0.247	0.389	0.249	0.46

TABLE V: F1-scores on malware classification task.

Model	Malware Class									Average
	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak	
IMCFN [23]	0.529	0.8	0.986	0.474	0.026	0.493	0.855	0.711	0.474	0.594
Malconv2 [21]	0.751	0.941	0.985	0.334	0.026	0.587	0.74	0.748	0.701	0.646
PROTST ₀	0.596	0.73	0.855	0.251	0	0.223	0.011	0.615	0.434	0.413
PROTST	0.824	0.948	0.986	0.837	0.182	0.795	0.871	0.862	0.905	0.801

TABLE VI: F1-scores on function signature prediction: (Left) argument count prediction and (Right) return type prediction

Model	Number of Arguments Class							Average
	0	1	2	3	4	5	others	
EKLAVYA [17]	0.149	0.321	0.325	0.282	0.585	0.121	0.659	0.349
Palmtree [10]	0.431	0.475	0.393	0.266	0.117	0.141	0.557	0.34
PROTST ₀	0.321	0.541	0.421	0.313	0.601	0.055	0.826	0.439
PROTST	0.853	0.84	0.755	0.737	0.783	0.436	0.94	0.763

Model	Return Type Class						Average
	int	char	void	double	bool	others	
EKLAVYA [17]	0.074	0.035	0.779	0	0.065	0.407	0.226
Palmtree [10]	0.291	0.102	0.758	0.511	0.202	0.248	0.352
PROTST ₀	0.015	0	0.78	0.075	0	0.465	0.223
PROTST	0.549	0.495	0.876	0.694	0.423	0.724	0.626

TABLE VII: F1-scores on function name prediction.

Model	Average
Asm2vec [36]	0.604
Symmlm [4]	0.778
PROTST ₀	0.698
PROTST	0.835

our more challenging fine-tuning setting. This suggests that the domain-specific knowledge incorporated by PalmTree through context window prediction (CWP) and def-use prediction (DUP) tasks is less effective than the knowledge transfer achieved with PROTST’s teacher-student paradigm. Furthermore, PROTST₀, performs competitively, matching EKLAVYA’s performance in return-type prediction (around 22% for both) and surpassing EKLAVYA and Palmtree in number of arguments prediction (43.9% vs. 34.9% and 34.0%). Yet, PROTST₀ fails to distinguish certain less frequent return-type classes than EKLAVYA, particularly for bool and char.

6) *Function Name Prediction*: The function name prediction task involves assigning human-readable names to functions within a binary file. We evaluate PROTST against two established methods: Symmlm [4], a transformer-based model that leverages context-sensitive, execution-aware code embeddings derived from assembly code, and Asm2vec [36], which employs random walks on the CFG to sample instruction sequences, and then uses the PV-DM [37] model for joint learning of function embedding. We note that function name prediction involves assigning names from a vocabulary of human-readable words; thus, we report the micro-F1 score for this task.

As shown in Table VII, PROTST achieves the highest average F1-score of 83.5%, outperforming both Asm2vec (60.4%) and Symmlm (77.8%). While Symmlm incorporates contextual information as multi-modal inputs, including execution traces

and call graph embeddings, in our setting with a limited fine-tuning dataset and demanding train-test split, it does not achieve the same level of performance as PROTST, which benefits from knowledge transfer from earlier tasks. Notably, PROTST₀ performs competitively, achieving an F1-score of 69.8%, surpassing Asm2vec [36]. This further demonstrates the value of pre-trained knowledge embedded in PROTST.

7) *Function Similarity Detection*: The function similarity detection task involves querying a pool of function embeddings to identify the most similar function to a given binary function. We evaluate PROTST against two established methods, Asm2vec [36] and jTrans [8], a transformer-based model that embeds control flow graphs with assembly code as input. We use MRR metrics for different function pool sizes (32 and 10K), as shown in Table VIII.

Our analysis reveals that the function similarity detection task benefits from a richer vocabulary, granting jTrans an inherent advantage due to its use of assembly code (over 10K tokens [9]) compared to PROTST’s raw byte representation (256 tokens). This advantage is evident for larger pool sizes. However, even with this limitation, PROTST achieves competitive results, particularly for smaller function pools (32 or fewer). We observe that PROTST shows a higher MRR for small pool sizes in most optimization settings. For larger pool sizes, the performance gap between PROTST and jTrans narrows to a 2% difference in average MRR. These results show the effectiveness of the teacher-student learning approach, where transferred knowledge (boundary and function-level knowledge) improves the function similarity detection accuracy. Furthermore, despite Asm2vec directly incorporating function CFG information into its model, PROTST₀ surpasses in MRR across all pool sizes, which shows the benefit of pre-training.

TABLE VIII: MRR on function similarity detection for pool sizes 32 and 10k.

Model	MRR						Average
	O0,O3	O1,O3	O2,O3	O0,O _s	O1,O _s	O2,O _s	
Asm2vec [36]	0.212/0.011	0.429/0.122	0.599/0.296	0.215/0.009	0.414/0.128	0.461/0.157	0.422/0.121
Jtrans [8]	0.762/ 0.267	0.911/0.611	0.976 /0.762	0.815/ 0.323	0.926/ 0.631	0.934/ 0.625	0.887/ 0.536
ProTST ₀	0.537/0.13	0.857/0.359	0.948/0.672	0.574/0.146	0.823/0.326	0.852/0.389	0.766/0.337
ProTST	0.796 /0.218	0.943 / 0.619	0.973/ 0.792	0.84 /0.256	0.944 /0.592	0.941 /0.614	0.906 /0.515

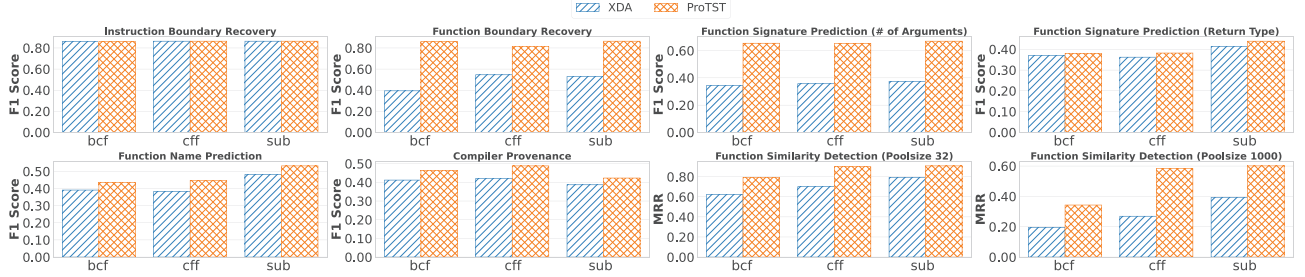


Fig. 3: Comparative performance of ProTST and XDA on various binary analysis tasks under different obfuscation methods.

B. RQ2: Generalization of ProTST

In Section V-A, we demonstrate the effectiveness of ProTST in various binary analysis tasks and compare it with the state-of-the-art methods. We now assess its generalization capabilities by evaluating its effectiveness under diverse code obfuscation techniques. We benchmark ProTST against XDA [3] to highlight the distinct advantages of the teacher-student learning paradigm. XDA, a traditional two-stage training approach, serves as a suitable baseline since it operates directly on the raw byte sequences.

Code Obfuscation. To assess the effectiveness of ProTST against code manipulation techniques, we evaluate its performance on obfuscated binaries. We used the llvm-obfuscator tool [38] to obfuscate a set of 51 popular open-source software projects (including binutils, curl, and gzip) with three distinct methods: bogus control flow (bcf), instruction substitution (sub), and control flow flattening (cff).

The obfuscation process required several modifications to our initial evaluation setup. First, since only a single obfuscator compiler (llvm-obfuscator) was used, the compiler prediction aspect of the compiler provenance task was omitted, focusing solely on predicting the optimization level. Second, applying obfuscation methods resulted in insufficient complete binary pairs (covering all optimization levels from O0 to Os) for the function similarity detection task with a pool size of 10K. Therefore, we report results using a smaller pool size of 1K to ensure sufficient data for robust evaluation. Third, due to the absence of corresponding source code in malware datasets, which is required by llvm-obfuscator, malware classification was excluded from this evaluation. We leveraged the binary knowledge learned from the pre-trained models in ProTST, without any pre-training on obfuscated binaries, and performed fine-tuning on the obfuscated data following the strategy outlined in Section IV-A.

Figure 3 summarizes the results. Our findings align with those from the optimization-level experiments. We observe that

ProTST outperforms XDA, achieving an average improvement of 16.6% across all obfuscation methods and tasks. This is a notable improvement considering that ProTST was not pre-trained on any obfuscated binaries. We postulate that explicitly training the model on obfuscated code at the pre-training stage could further improve its robustness and effectiveness against such code manipulation techniques.

C. RQ3: Ablation Study on Knowledge Transfer

We investigate the impact of the staged design in ProTST’s Binary Knowledge Accumulation (BKA) module, focusing on how each preceding task influences the performance of downstream tasks. To investigate this, we conduct ablation experiments to evaluate two configurations of the BKA stage with altered task orders (C-A and C-B), as depicted in Figure 4.

Table IX summarizes the ablation study results. Due to the similarity between C-A and C-B, only the different settings and their corresponding results are reported for C-B. We exclude function name prediction and malware classification from evaluation as teacher modules since they are leaf tasks in both configurations. Similarly, function similarity detection, a leaf task in C-A, does not serve as a teacher in that configuration.

In summary, as detailed below, our results demonstrate a consistent trend: Each task’s performance improves by incorporating prior knowledge from the preceding teacher tasks. As we progress from a lower ID (less prior knowledge) to an adjacent higher ID (more prior knowledge), we observe a minimum performance gain of 2%. Notably, comparing the best results (bolded in the table) for each task with the traditional two-stage training baseline (ID-1) reveals a consistent improvement of an average of 14.8% with at least 5% across all downstream tasks.

Impact of Task Order. We observe that C-B surpasses C-A in function signature prediction, demonstrating notably higher performance for both the prediction of the number of arguments (76.3% in ID-6 compared to 62.6% in ID-3) and return type

TABLE IX: Analyzing task effectiveness in our design choices. Each row corresponds to a model pre-trained with a specific combination of task modules listed in the “Teacher Modules” column. Row $ID - 0$ serves as the baseline without pre-training and row $ID - 1$ represents the traditional two-stage framework with only MLM pre-training. A dash (-) indicates no result for a specific student task under a particular combination of teacher modules.

ID	Config.	Teacher Modules							Student Tasks									
		MLM	Inst Boun	Func Boun	Func Sim	Func Sig	Comp Prov	Inst Boun	Func Boun	Func Sig	Count	Func Sig	Type	Func Name	Func Sim 32	Func Sim 10000	Comp Prov	Mal CLS
0	A							0.871	0.478		0.529	0.495	0.641	0.768	0.257	0.368	0.714	
1		✓						0.884	0.492	0.538	0.556	0.693	0.791	0.282	0.406	0.73		
2		✓	✓					-	0.915	0.62	0.587	0.788	0.816	0.29	0.449	0.743		
3		✓	✓	✓				-	-	0.626	0.613	0.805	0.827	0.332	0.46	0.787		
4		✓	✓	✓			✓	-	-	-	-	0.814	0.865	0.393	-	-		
5		✓	✓	✓	✓			-	-	-	-	-	-	-	-	0.801		
6	B	✓	✓	✓	✓	✓		-	-	0.763	0.626	0.81	-	-	-	-		
7		✓	✓	✓	✓	✓	✓	-	-	-	-	0.835	0.906	0.515	-	-		

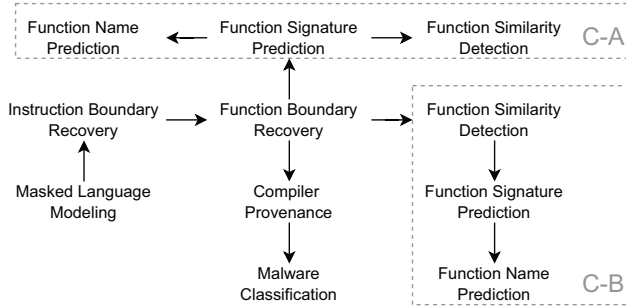


Fig. 4: Different knowledge transfer configs (C-A and C-B).

(62.6% in ID-6 compared to 61.3% in ID-3). Similarly, function name prediction also benefits from the task order in C-B, reaching 83.5% accuracy (ID-7) compared to 81.4% in C-A (ID-4). This improvement is attributable to the inclusion of function similarity detection as an additional teacher task in C-B, which provides the model with valuable knowledge beneficial for the downstream tasks.

However, C-B presents a trade-off. Although it excels in function signature and name prediction, its performance in function similarity detection is marginally diminished compared to C-A. In C-A, function similarity detection achieves 86.5% accuracy for a function pool size of 32 (ID-4), and 39.3% for a pool size of 10K (ID-4). In contrast, when placed earlier in the BKA hierarchy in C-B (ID-3), it receives less knowledge accumulated from previous tasks, resulting in slightly lower performance (82.7% for a pool size of 32 and 33.2% for a pool size of 10K). This finding underscores the significance of task order, as tasks positioned later in the hierarchy can leverage accumulated knowledge from teacher tasks and lead to improved performance.

VI. DISCUSSION AND LIMITATIONS

In this work, we opted for raw byte input rather than assembly code due to the latter’s substantially larger vocabulary [9], which would significantly increase the convergence time in our multi-stage pre-training framework; however, the core knowledge transfer technique of PROTST remains applicable to assembly code by substituting our raw-byte backbone model with an assembly-based one from previous studies [4], [5],

[8], [9], [10]. This adaptability enables the incorporation of additional binary analysis tasks, including those based on assembly-level input, into the PROTST’s learning process, potentially enhancing its overall capabilities. Future work will explore tasks such as vulnerability search, indirect call recognition, and memory dependency analysis within PROTST.

Although we primarily focus on the x86 architecture, PROTST is readily applicable to other instruction sets, such as ARM and MIPS, particularly with datasets like [33] that include binaries for these architectures. PROTST requires only the ground truth of binary tasks, which can be readily obtained from labeled datasets or through manual extraction. Future work will explore the application of PROTST to these architectures.

VII. CONCLUSION

We introduce PROTST¹, a Progressive Teacher-Student Binary Analysis framework specifically designed to improve the accuracy of binary analysis tasks. Unlike traditional two-stage training approaches, PROTST employs a hierarchical tree structure that facilitates a progressive knowledge transfer from fundamental to more specialized tasks. This hierarchical design ensures a natural and logical flow of information, where foundational tasks establish a robust base for more complex tasks. This progressive approach minimizes the reliance on external tools and avoids the tedious processes associated with reverse engineering and feature extraction, thereby simplifying the incorporation of binary code knowledge. Our extensive evaluations highlight the efficacy of PROTST in seven different binary analysis tasks. Our method represents a potentially important step forward in binary analysis tasks and opens up new avenues for the application of knowledge transfer.

ACKNOWLEDGMENT

The authors thank anonymous reviewers for their valuable suggestions. This work was supported by NSF under Grant IIS-2229876. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

¹The development and evaluation of PROTST was a complex endeavor. An extended version of this paper [39] provides additional details.

REFERENCES

- [1] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," in *Association for Computational Linguistics (ACL)*, 2019.
- [2] N. Jiang, C. Wang, K. Liu, X. Xu, L. Tan, and X. Zhang, "Nova⁺: Generative language models for binaries," *arXiv preprint arXiv:2311.13721*, 2023.
- [3] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," in *Network and Distributed System Security Symposium (NDSS)*, 2021.
- [4] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symml: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.
- [5] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *IEEE Transactions on Software Engineering (TSE)*, 2022.
- [6] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: Fine-grained type recovery from binaries using generative state modeling," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.
- [7] X. He, S. Wang, Y. Xing, P. Feng, H. Wang, Q. Li, S. Chen, and K. Sun, "Binprov: Binary code provenance identification without disassembly," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2022.
- [8] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "Jtrans: Jump-aware transformer for binary code similarity detection," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.
- [9] W. Zhu, H. Wang, Y. Zhou, J. Wang, Z. Sha, Z. Gao, and C. Zhang, "ktrans: Knowledge-aware transformer for binary code embedding," *arXiv preprint arXiv:2308.12659*, 2023.
- [10] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.
- [11] K. Pei, D. She, M. Wang, S. Geng, Z. Xuan, Y. David, J. Yang, S. Jana, and B. Ray, "Neudep: neural binary memory dependence analysis," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.
- [12] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *USENIX Security*, 2014.
- [13] D. Lopez-Paz, L. Bottou, B. Schölkopf, and V. Vapnik, "Unifying distillation and privileged information," *International Conference on Learning Representations (ICLR)*, 2015.
- [14] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.
- [15] "Protst supplementary," <https://github.com/purseclab/ProTST>, 2024, [Online; accessed 15-Dec-2024].
- [16] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2019.
- [17] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *USENIX Security*, 2017.
- [18] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *USENIX Security*, 2015.
- [19] J. Schmidhuber, S. Hochreiter *et al.*, "Long short-term memory," *Neural Comput.*, 1997.
- [20] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *NIPS Workshop on Deep Learning*, 2014.
- [21] E. Raff, W. Fleshman, R. Zak, H. S. Anderson, B. Filar, and M. McLean, "Classifying sequences of extreme length with constant memory applied to malware detection," in *AAAI Conference on Artificial Intelligence*, 2021.
- [22] Y. Otsubo, A. Otsuka, M. Mimura, and T. Sakaki, "o-glasses: Visualizing x86 code from binary using a 1d-cnn," *IEEE Access*, 2020.
- [23] D. Vasan, M. Alazab, S. Wassan, H. Naeem, B. Safaei, and Q. Zheng, "Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture," *Computer Networks*, 2020.
- [24] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [25] S. Yu, Y. Qu, X. Hu, and H. Yin, "Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *USENIX Security*, 2022.
- [26] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, R. Baldoni *et al.*, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *NDSS Workshop on Binary Analysis Research (BAR)*, 2019.
- [27] F. Artuso, M. Mormando, G. A. Di Luna, and L. Querzoni, "Binbert: Binary code understanding with a fine-tunable and execution-aware transformer," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2024.
- [28] D. Gibert, "Machine learning for windows malware detection and classification: Methods, challenges and ongoing research," *arXiv preprint arXiv:2404.18541*, 2024.
- [29] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Computing Surveys*, 2023.
- [30] "Gnu binutils," <https://www.gnu.org/software/binutils/>, 2023, [Online; accessed 25-Jun-2024].
- [31] "Spec cpu2017 benchmark suite," <https://www.spec.org/cpu2017/>, 2017, [Online; accessed 25-Jun-2024].
- [32] "Spec cpu2006 benchmark suite," <https://www.spec.org/cpu2006/>, 2006, [Online; accessed 25-Jun-2024].
- [33] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering (TSE)*, 2022.
- [34] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," in *arXiv 1802.10135*, 2018.
- [35] Y. Xian, B. Schiele, and Z. Akata, "Zero-shot learning-the good, the bad and the ugly," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
- [36] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [37] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning (ICML)*, 2014.
- [38] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm—software protection for the masses," in *ACM/IEEE ICSE Workshop on Software Protection*, 2015.
- [39] H. Lu, H. Cai, Y. Liang, A. Bianchi, and Z. B. Celik, "A progressive transformer for unifying binary code embedding and knowledge transfer (extended paper)," in *arXiv:2412.11177*, 2025.