Extended version of A Progressive Transformer for Unifying Binary Code Embedding and Knowledge Transfer
(Accepted to IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2025)

A Progressive Transformer for Unifying Binary Code Embedding and Knowledge Transfer

Hanxiao Lu[1], Hongyu Cai[2], Yiming Liang[2], Antonio Bianchi[2], Z. Berkay Celik[2]
[1]*Columbia University, hl3424@columbia.edu*
[2]*Purdue University, {hongyu, liang328, antoniob, zcelik}@purdue.edu*

***Abstract*—Language model approaches have recently been integrated into 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. However, this approach involves complex feature engineering, a cumbersome and time-consuming process that can introduce predictive uncertainty when dealing with stripped or obfuscated code, leading to a performance drop. In this paper, 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, and the results show that PROTST yields an average validation score (F1, MRR, and Recall@1) improvement of 14.8% compared to traditional two-stage training and an average validation score of 10.7% compared to multimodal two-stage frameworks.**

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. Fig. 1(a) shows two recent approaches, XDA [3] and Binprov [7], which 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], as shown in Fig 1(b), 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.

We evaluate PROTST on seven binary analysis tasks with diverse datasets. We then compare its performance with state-of-the-art methods in the same experimental setting. Our experiment reveals significant advantages of the teacher-student paradigm compared to a traditional two-stage training architecture. When the teacher-student approach is introduced, we observe an average 14.8% with at least 5% improvement in validation score (F1, MRR, and Recall@1) in all tasks and an average 3X times faster in convergence. Furthermore, we demonstrate improvements in binaries compiled using various optimization settings and dealing with obfuscated data. Compared with multi-modal two-stage frameworks, PROTST outperforms these models by an average of 10.7% validation score in all tasks.

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 consisting of three key components: (1) an embedding module that transforms raw binary code into a high-dimensional representation, (2) a transformer backbone model that captures code features, and (3) task-specific heads designed for various objectives.
- We evaluate PROTST on seven diverse binary analysis tasks. Our results demonstrate that PROTST yields an average validation score improvement of 14.8% compared to traditional two-stage training and an average validation score of 10.7% compared to multi-modal two-stage frameworks.

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 a lower-dimensional space, allowing these embeddings to be used for various binary analysis tasks. We focus on deep learning-based approaches to BCE, specifically those that could operate directly on raw-byte sequences. Tasks such as vulnerability search [25], [26], [27], memory dependency analysis [28], [11], variable type recovery [6], indirect call recognition [29], and binary code comprehension [30], [31] require assembly or higher-level pseudo-code for analysis and are therefore not within the scope of this paper. Below, we present BCE approaches by grouping them into learning-based and transformer-based. Table I compares key characteristics of recent works using these approaches and PROTST.

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 [23], EKLAVYA [20], and Bi-RNN [19] use RNNs (e.g., LSTMs [32], GRUs [33]) for tasks such as code similarity detection, function signature prediction, and boundary detection. Malconv2 [17] and o-glasses [21] use 1-d CNNs to capture binary code embeddings for tasks such as malware classification and compiler provenance. Additionally, IMCFN [22] takes a different approach, using 2-dimensional CNNs to embed binaries as images specifically for malware classification. Techniques such as Gemini [34], DeepDi [18], and Structure2Vec [35] use GNNs to model binary code using graph representations, e.g., CFGs, DFGs.

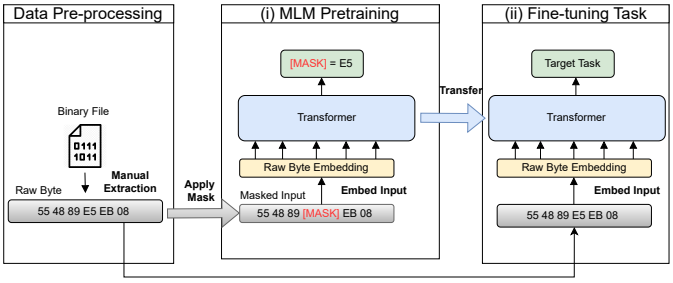
Current learning-based methods, however, often neglect the nuances of individual instruction formats and complexities. Their focus on static analysis also limits their ability to incorporate broader contextual information essential for comprehensive binary semantic understanding.

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] (Figure 1(a)) 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 enhance understanding (Figure 1(b)). 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] for tasks that include function signature recovery and function similarity detection. A line of work, such as Trex [5] and BinBert [16], model the dynamic

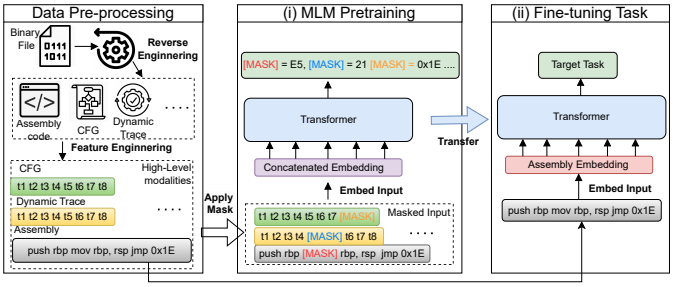
TABLE I: Properties of existing Binary Code Embedding (BCE) approaches and their comparison with ProTST.

		Transformer-based								RNN/CNN/GNN-based								
		XDA [3]	Binprov [7]	Palmtree [10]	jTrans [8]	kTrans [9]	Trex [5]	Symlm [4]	BinBert [16]	Malconv2 [17]	DeepDi [18]	Bi-RNN [19]	EKLAVYA [20]	o-glasses [21]	IMCFN [22]	SAFE [23]	Gemini [24]	ProTST ★
Input Modality	Raw Bytes	●	●	·	·	·	·	·	·	●	·	●	·	●	●	·	●	●
	Assembly	·	·	●	●	●	●	●	●	·	●	·	●	·	·	●	●	·
	CFG	·	·	●	●	·	·	·	·	·	·	·	·	·	·	·	●	·
	DFG	·	·	●	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	Register Info	·	·	·	·	●	·	·	·	·	·	·	·	·	·	·	·	·
	Operand Info	·	·	·	·	●	·	·	·	·	·	·	·	·	·	·	·	·
	Caller/Callee Info	·	·	·	·	·	·	●	·	·	·	·	·	·	·	·	·	·
	Dynamic Behavior	·	·	·	·	·	●	●	●	·	·	·	·	·	·	·	·	·
Binary Task	Disassembly	●	·	·	·	·	·	·	·	·	●	●	·	·	·	·	·	●
	Compiler Provenance	·	●	·	·	·	·	·	●	·	·	·	·	●	·	·	·	●
	Malware Classification	·	·	·	·	·	·	·	·	●	·	·	·	·	●	·	·	●
	Function Signature	·	·	●	·	●	·	·	·	·	·	·	●	·	·	·	·	●
	Function Name	·	·	·	·	·	·	●	·	·	·	·	·	·	·	·	·	●
	Function Similarity	·	·	●	●	●	●	·	●	·	·	·	·	·	·	●	●	●
Equipped with Binary code Knowledge		·	·	◑	◑	◑	◑	◑	◑	·	◑	·	·	·	·	·	◑	●
No Feature Engineering		●	●	·	·	·	·	·	·	●	·	●	●	●	●	●	·	●

● This mark denotes the feature is fully implemented in the model.
◑ This mark denotes that the model has a partial implementation of this feature.



(a) Traditional two-stage architecture.



(b) Two-stage framework with multi-modality.

Fig. 1: (a) Traditional two-stage training on static code. (b) Two-stage training with high-level modalities (e.g., CFG, DFG, and execution traces).

behavior of binary code within the transformer architecture for tasks, e.g., function similarity detection. Other methods, kTrans [9] (using register and operand information) and Symlm [4] (leveraging caller/callee information), demonstrate the potential of incorporating specific details for tasks such as function name prediction and function similarity detection.

Existing transformer-based methods for binary analysis, however, face two challenges. First, the extraction of features requires specialized knowledge per architecture, limiting their use in diverse datasets. Popular reverse engineering tools face difficulties with obfuscated code and misinterpret functions/instructions, introducing data errors that degrade model performance. Second, indiscriminately incorporating high-level information during pre-training can negatively impact model performance, especially if the information isn't closely relevant to the specific tasks the model will be fine-tuned for later.

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 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 feature and reverse engineering, or explicit knowledge of the binary code structure.

Motivation and Approach Overview. Figure 2 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 (❶) to understand the syntax of the language at the byte level, and (b) a novel Binary Knowledge Accumulation (BKA) stage (❷-❽) 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 (❷-❸), function-level knowledge (❹-❻), and file-level knowledge (❼-❽).

The initial MLM stage (❶) 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 (❷), leveraging the foundational knowledge from the MLM stage to identify boundaries between individual instructions. This crucial step enables the model to group bytes

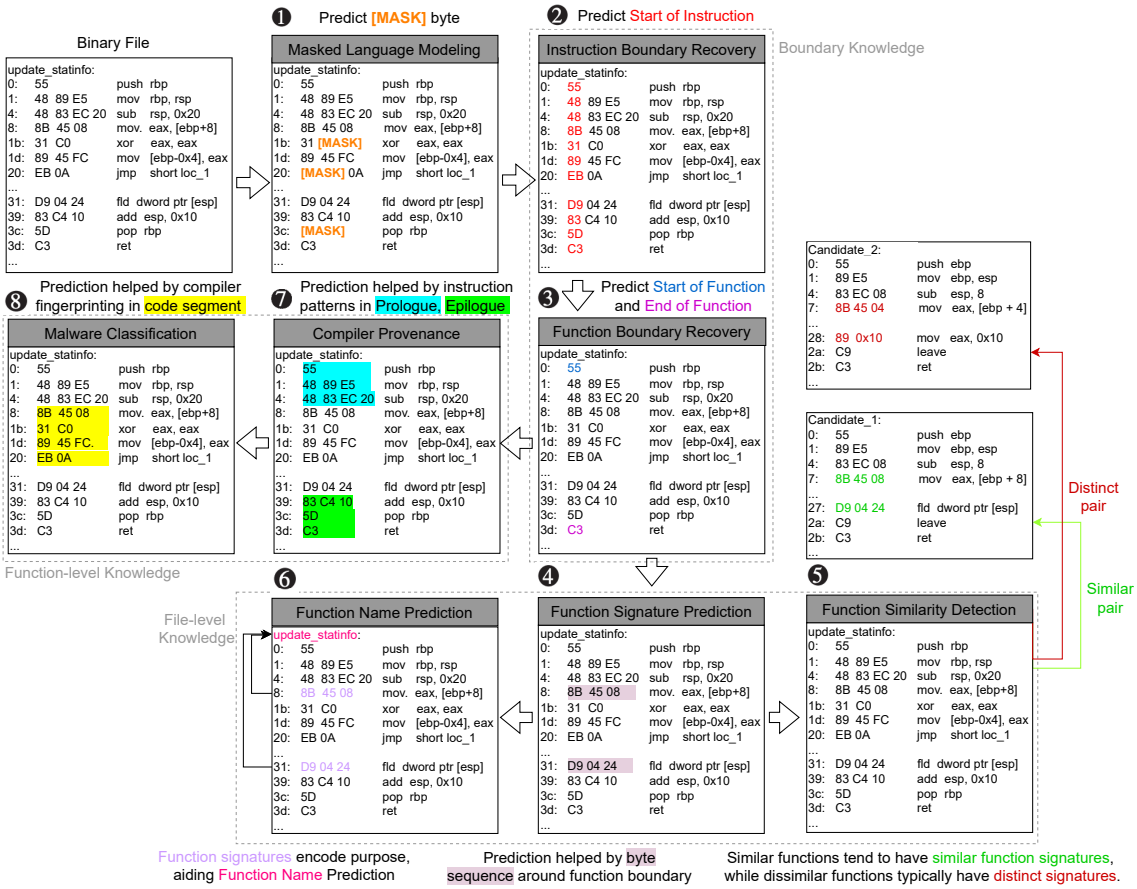


Fig. 2: The progressive *teacher-student* learning of ProTST. Tasks are hierarchically structured to leverage foundational knowledge for more complex tasks. Each task employs a transformer, with model weights serving as interfaces between adjacent nodes. The system operates solely on the raw byte sequence (address and assembly are shown for illustration only).

into meaningful instruction sequences, similar to parsing words from sentences. Building on this foundation, function boundary recovery (❸) 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 (❹), which act as 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 (❺). 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 (❻). 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 (❼) and malware classification (❽). 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 is crucial. The design of PROTST assumes that malware from the same family is often compiled using the same (or a similar) compiler [36]. Consequently, identifying the compiler used can be instrumental in tracing malware's origins and family.

A. Model Architecture

Figure 3 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: $x = \{0x00, ..., 0xff\}^n$. 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. In addition to the possible 256 byte values, we add 5 special tokens to the input vocabulary, including [CLS] at the beginning and [SEP] at the end. [PAD] tokens are appended to the end of the token sequence to ensure equal length for each sequence. Token sequences that exceed the maximum length limit are truncated. For tokens not found in the vocabulary, we uniformly represent them with the special [UNK] token. [Mask] tokens are applied to the input byte token to perform MLM pre-training. Notably, we do not impose constraints on the input sequence length n. This allows flexibility, enabling byte sequences to span the entire binary program or focus on specific subsets of instructions within a single binary.

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:

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

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 tokens (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 $X = [x_1, x_2, ..., x_n]$ 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:

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

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 $t \in \{1, 2, ..., T\}$ employing a dedicated instance of the RoBERTa transformer, denoted as M_t. The knowledge transfer process proceeds sequentially through this task chain. More specifically, we consider Θ_t to represent 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:

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

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:

$$\Theta_t^{init} = \Theta_{t-1}^* \tag{4}$$

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

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

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 [37]. 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).

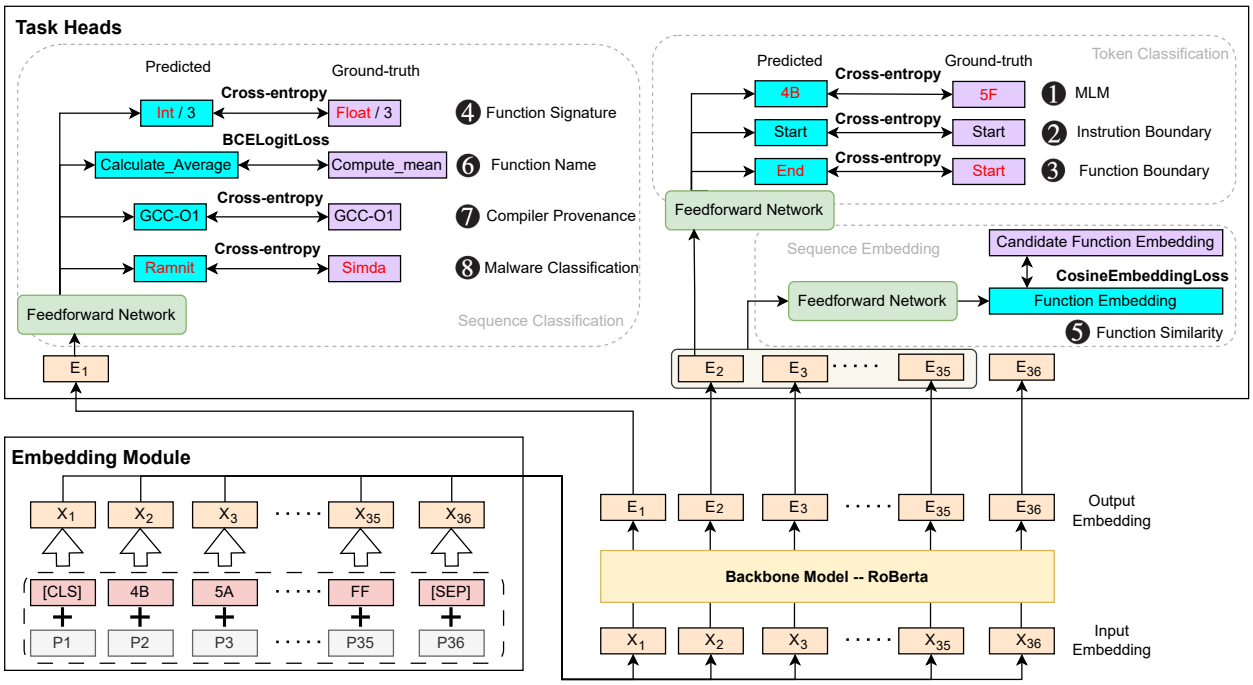


Fig. 3: The model architecture of PROTST

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.

Masked Language Modeling Head. The MLM head (❶ in Figure 3) operates during the initial pre-training stage. It is designed to predict masked bytes within the binary code by leveraging the context from surrounding tokens. This task encourages the model to develop a fundamental understanding of the structure and semantics inherent in the raw byte sequence, akin to learning the vocabulary and grammar of a new language.

We adopt a configurable MLM strategy [3], where we randomly mask a proportion p_{mask} of the input bytes. Of these masked bytes, a fraction p_{replace} are replaced with the special [MASK] token, while the remaining $1 - p_{\text{replace}}$ are replaced with random bytes from the vocabulary {0x00, ..., 0xff}.

Formally, let x denote the original input byte sequence and m^x the indices of the masked tokens. The masked input sequence x^{MLM} can be expressed as:

$$x^{MLM} = REPLACE(x, m^x, [MASK]) \tag{6}$$

The objective of the MLM head is to reconstruct the original masked bytes, formulated as the following loss function:

$$\mathcal{L}_{MLM} = -\sum_{i \in m^x} \log P(x_i \mid x^{MLM}) \tag{7}$$

This objective function drives the model to maximize the probability of reconstructing the original masked bytes to learn meaningful data representations.

Binary Task Heads. Following the MLM pre-training, 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.

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 ❹, ❻-❽). It operates on the final hidden state $h_{[CLS]}$ corresponding to the [CLS] token produced by the backbone model.

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 ❷-❸) 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 Embedding Head. We use this head for the task of function similarity detection (❺). It generates a fixed-length embedding representation e for an entire byte sequence x by aggregating (e.g., averaging) the output embeddings $h_1, h_2, ..., h_n$ 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.

The choice of loss function depends on the specific task. For multi-class, multi-label classification tasks such as function name prediction, we employ the binary cross-entropy loss with logits (BCELogit Loss). For function similarity detection, which involves embedding comparison, we use the Cosine Embedding Loss. For all other standard multi-class classification tasks, the Cross-Entropy Loss is employed. Detailed information on the

ground truth for each task and the specific configuration of each task head can be found in Appendix A.

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 II and their details are given in Appendix B.

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 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 p_{replace} 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.

TABLE II: Binary datasets used to evaluate PROTST.

Dataset	Data Size	Task
Binutils [38]	1 project	Masked Language Modeling
SPEC CPU [39], [40]	2.01G bytes	Inst./Func. Boundary Prediction
BAP [12]	345M bytes	Func. Boundary Prediction
BinKit [41]	75M functions	Compiler Prov./Func. Signature Prediction
Symlm [4]	1.44M functions	Func. Name Prediction
Binarycorp-3M [8]	404K functions	Func. Similarity Detection
BIG2015 [42]	10860 files	Malware Classification

Following recent work [3], [4], [6], the primary metric used to evaluate most tasks in PROTST is the F1 score. To account for potential class imbalances that might skew performance assessments, we specifically 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. Then we average the F1-scores across all classes, which yields the macro-F1 score:

$$\text{Macro-F1} = \frac{1}{|C|} \sum_{c \in C} F1_c \tag{8}$$

where $|C|$ denotes the total number of classes, and $F1_c$ represents the F1-score for class c. This ensures that all classes contribute equally to the overall evaluation regardless of their frequency in the dataset.

For the task of function similarity detection, we adopt the Mean Reciprocal Rank (MRR) and Recall@k metrics 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:

$$MRR = \frac{1}{|Q|} \sum_{q_i \in Q} \frac{1}{\text{rank}(q_i^{gt})} \tag{9}$$

where $|Q|$ is the total number of query functions in the evaluation set, q_i represents a query function, q_i^{gt} denotes its ground truth counterpart, and $\text{rank}(q_i^{gt})$ indicates the position of the ground truth function in the ranked list returned by the model for query q_i. A lower rank signifies a better result.

Recall@k, as a complementary metric, measures the proportion of queries in which the ground truth function is included within the top k results retrieved by the model:

$$Recall@k = \frac{1}{|Q|} \sum_{q_i \in Q} \mathbb{I}(\text{rank}(q_i^{gt}) \leq k) \tag{10}$$

where $\mathbb{I}(\cdot)$ is an indicator function that equals 1 if the condition inside the parentheses is true and 0 otherwise.

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 the similarity at varying sizes (32 and 10K functions). To fully assess the PROTST's performance, we report metrics for each subtask.

TABLE III: Results on instruction boundary recovery.

Model	Class		Average
	Start	Middle	
Bi-RNN [19]	0.443	0.885	0.664
DeepDi [18]	0.765	**0.998**	0.881
PROTST$_0$	0.61	0.906	0.758
PROTST	**0.827**	0.941	**0.884**

TABLE IV: Results on function boundary recovery.

Model	Class			Average
	Middle	Start	End	
Bi-RNN [19]	0.997	0	0	0.332
DeepDi [18]	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**

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 varying binary optimization levels and code obfuscation techniques?

RQ3 How does the order of tasks in the BKA stage influence knowledge transfer and impact overall performance?

RQ4 What is the computational efficiency of PROTST compared to alternative approaches, and how does it scale with task complexity and dataset size?

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 models for each class within the dataset.

To further investigate the efficacy of the teacher-student paradigm, we introduce a variant model, PROTST$_0$. In contrast to PROTST, which leverages fine-tuning on student tasks to refine the transferred knowledge, PROTST$_0$ directly leverages the embeddings generated by the pre-trained teacher model. This approach aligns with the principles of zero-shot learning [43], in which a model is evaluated on unseen tasks without any task-specific adaptation. The performance of PROTST$_0$ is 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 by Shin et al. [19] and DeepDi [18], 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 III 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 recovery. Second, PROTST$_0$ 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 [19] and DeepDi [18]. 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 IV, 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, PROTST$_0$, shows a rudimentary ability to classify function boundaries and achieves F1-scores of 6.6% and 4.0% for 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

TABLE V: Results on compiler provenance.

Model	GCC						Clang						Average
	O0	O1	O2	O3	Os	Ofast	O0	O1	O2	O3	Os	Ofast	
o-glasses [21]	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 VI: Results on malware classification task.

Model	Malware Class									Average
	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak	
IMCFN [22]	0.529	0.8	0.986	0.474	0.026	0.493	0.855	0.711	0.474	0.594
Malconv2 [17]	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**

specific optimization level (O0, O1, . . . ,Ofast) used to generate a given binary file (see Appendix A for class descriptions). We evaluate the performance of PROTST against two established methods: O-glasses [21], 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 V 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 PROTST$_0$, 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 [22], which uses a 2D-CNN to extract image-like features from byte sequences, and Malconv2 [17], which employs a 1D-CNN to directly extract features from the raw 1D representation of the malware bytes.

As shown in Table VI, 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 PROTST$_0$ 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 [20], 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 VII summarizes the results. Our analysis reveals that 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 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 through PROTST's teacher-student paradigm. Furthermore, PROTST$_0$, 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%). However, PROTST$_0$ 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: Symlm [4], a transformer-based model that leverages context-sensitive, execution-aware code embeddings derived from assembly code, and Asm2vec [44], which employs random walks on the CFG to sample instruction sequences, and then utilizes the PV-DM [45] model for joint learning of function embedding. It is important to note that function name prediction involves assigning names from a vocabulary of human-readable words; therefore, we report the micro-F1 score for this task.

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

TABLE VII: Results 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 [20]	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 [20]	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 VIII: Results on function name prediction.

Model	Average
Asm2vec [44]	0.604
Symlm [4]	0.778
PROTST$_0$	0.698
PROTST	**0.835**

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, PROTST$_0$ performs competitively, achieving an F1-score of 69.8%, surpassing Asm2vec [44]. 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 [44] and jTrans [8], a transformer-based model that embeds control flow graphs with assembly code as input. We use MRR and Recall@1 metrics for different function pool sizes (32 and 10K), as shown in Table IX. In Figure 4, we also present Recall@1 results for various optimization combinations across a broader range of pool sizes (i.e., 2, 10, ..., 10K).

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 particularly 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 and Recall@1 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, PROTST$_0$ consistently surpasses in both MRR and Recall@1 across all pool sizes, which shows the benefit of pre-training.

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 in two key aspects: (1) PROTST's performance on binaries compiled with different optimization levels (e.g., O0, O1, O2), and (2) 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 as it operates directly on raw byte sequences.

Optimization Levels. To assess PROTST's ability to address variations in compiler optimization, we categorize binaries based on their optimization flags, ranging from minimal optimization (O0) to aggressive levels (e.g., Ofast). We note that specific optimization levels may vary depending on the compiler and dataset. Due to the absence of optimization information in the BIG2015 dataset, malware classification was excluded from this evaluation. We then evaluated the performance of both PROTST and XDA in these categorized datasets, with each optimization level (or optimization pair for function similarity detection) undergoing 100 fine-tuning epochs.

The results are presented in Figure 5. For instruction boundary recovery, as expected, the initial task in our teacher-student paradigm, both PROTST and XDA exhibit comparable performance across optimization levels. However, for subsequent tasks, PROTST shows significantly improved effectiveness compared to XDA, achieving an average performance improvement of 18.4% at various optimization levels. This substantial improvement is mainly due to the knowledge transfer with teacher-student learning process, which improves the model's ability to generalize to unseen optimization settings.

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 [46] 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.

TABLE IX: Results on function similarity detection for pool sizes 32 and 10k.

Model	MRR						Average
	O0,O3	O1,O3	O2,O3	O0,Os	O1,Os	O2,Os	
Asm2vec [44]	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

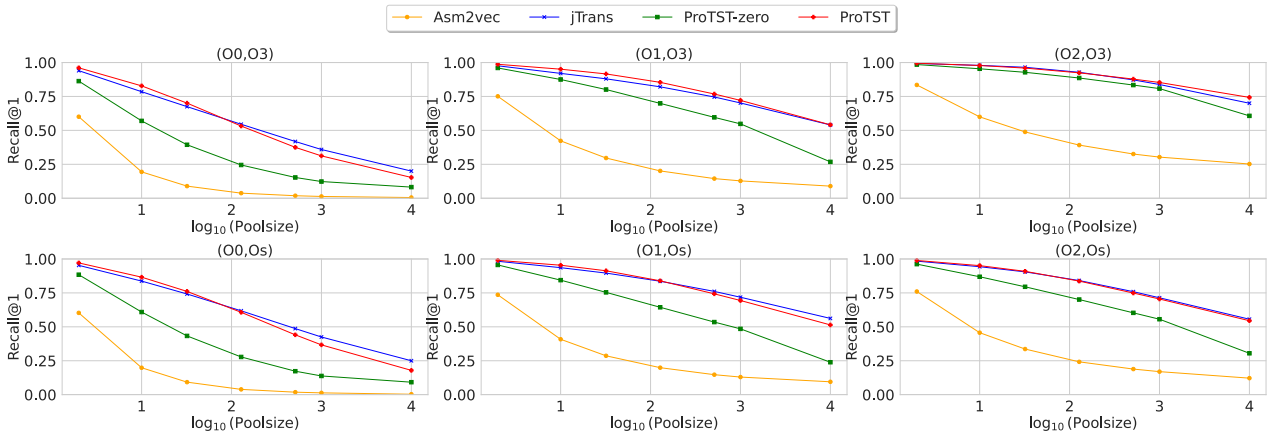


Fig. 4: The performance (Recall@1) of different models for binary code similarity detection with respect to pool size.

Fig. 5: Comparative performance of PROTST and XDA on various binary analysis tasks across different optimization levels.

Figure 6 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 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 7.

Table X summarizes the ablation study results. Due to the similarity between C-A and C-B, only the differing 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, 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 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 (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,

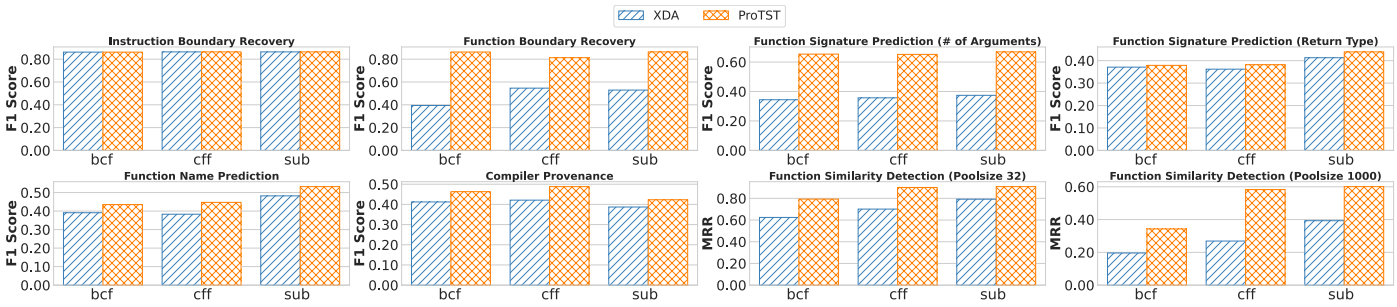


Fig. 6: Comparative performance of PROTST and XDA on various binary analysis tasks under different obfuscation methods.

TABLE X: 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								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	A	✓	✓					-	**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**	-	-

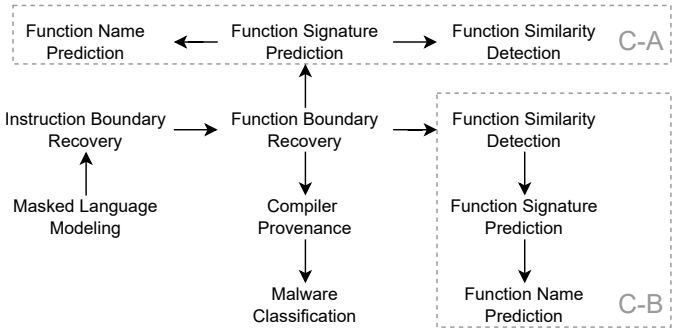


Fig. 7: Different knowledge transfer configs (C-A and C-B).

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.

Impact of Cyclical Transfer. We investigate cyclical knowledge transfer within the BKA module, where tasks iteratively refine each other's representations. Specifically, we show that one task can enhance the embeddings of another, subsequently improving the original task through this refined knowledge. We illustrate this using function similarity detection (ID-7 in C-B), where function signature prediction (ID-7) benefits from having function similarity detection as an additional teacher task compared to C-A (ID-4). The knowledge gained from function similarity detection during pre-training enhances function signature prediction accuracy, which is then transferred back to function similarity detection during fine-tuning. We ensure no data leakage by keeping the fine-tuning and pre-training data separate (Section IV-A).

Our results demonstrate significant improvements in function similarity detection when it serves as a teacher task for function signature prediction. ID-7 outperforms ID-4 (pool size 32: 90.6% vs. 86.5%; pool size 10K: 51.5% vs. 39.3%). This cyclical transfer strategy, applicable to any task configuration, opens avenues for diverse teacher-student combinations, potentially enhancing the performance of all involved tasks.

D. RQ4: Pre-training Effectiveness

To evaluate PROTST's efficiency in adapting to new tasks with limited data and fine-tuning epochs, we perform two experiments. First, we analyze the number of fine-tuning epochs required for PROTST to achieve the desired performance on validation data. This enables us to evaluate how effectively the model can adapt to new tasks with limited fine-tuning. Second, we examine the impact of limiting the number of training samples used in the teacher's tasks during pre-training. This simulates resource constraints and assesses the generalizability of PROTST to such limitations. We benchmark PROTST against XDA [3], a model that lacks progressive knowledge transfer.

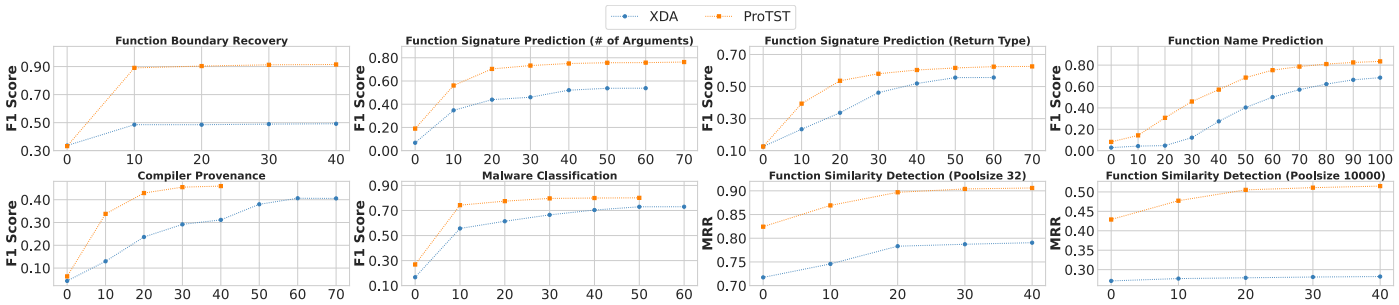


Fig. 8: Convergence analysis of validation scores between ProTST and XDA.

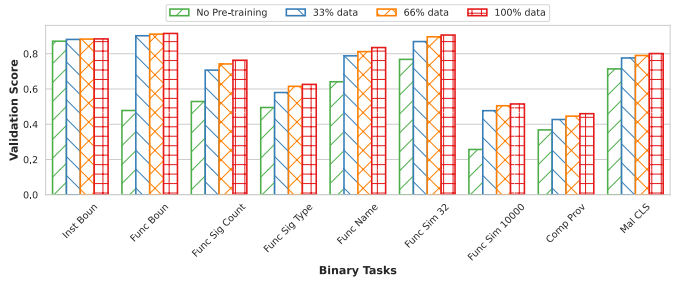


Fig. 9: The results of data scaling experiments.

Fine-tuning Epochs. We assess whether ProTST requires fewer fine-tuning epochs than XDA to exceed a chosen performance threshold on the validation data. A lower number of epochs indicates faster convergence and suggests that the pre-trained model is equipped with richer knowledge, which allows for a quicker adaptation to new tasks. We note that the instruction boundary recovery task, being the first task, is expected to exhibit the same convergence speed. Therefore, we exclude this task from the fine-tuning epoch comparison.

Figure 8 presents the convergence analysis of the validation scores between ProTST and XDA. ProTST requires fewer fine-tuning epochs to exceed a given threshold than XDA in all downstream tasks. For example, in the compiler provenance task, XDA requires 30 epochs to reach an F-1 score of 0.3, while ProTST achieves this in approximately 10 epochs. Across most binary tasks, it shows an average convergence of two to three times faster than XDA to reach a specific performance threshold. This is because the BKA module accumulates different types of binary knowledge progressively, and the acquired knowledge base facilitates faster convergence during fine-tuning on downstream tasks.

Data Scaling. To investigate the impact of pre-training data size on the performance of ProTST, we evaluate four scenarios: The model is fine-tuned directly without any pre-training stage (S1), pre-trained using only 33% of the available data (S2), pre-trained using only 66% of the available data (S3), and pre-trained using the entire dataset (S4).

As shown in Figure 9, the model performance suffers significantly (with an average drop of 17.6%) when no pre-training is applied (S1). This observation highlights the importance of pre-training in equipping the model with foundational knowledge for effective downstream performance. Interestingly, we observe that the fine-tuned models achieve a similar validation score. Even when pre-trained with only 33% (S2) or 66% (S3) of the data, the decrease in the validation score compared to the entire dataset pre-training (S4) is minimal, staying within the range 4% for most binary tasks. This suggests that we can potentially achieve similar performance levels while using a reduced amount of pre-training data; this demonstrates the efficiency of ProTST's pre-training process and its ability to leverage a smaller dataset.

VI. Discussion and Limitations

Due to computational constraints, we opted for raw byte input, as assembly code possesses a significantly larger vocabulary [9], which would lead to a significantly longer 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.

The hierarchical task order in ProTST is currently manually determined based on the logical flow of binary knowledge. However, due to computational constraints, the optimal task ordering for ProTST has not been fully explored. A promising future direction involves leveraging curriculum learning [47] to extend PROTST to new tasks and optimize the task order in a more systematic manner. Additionally, to accommodate an increasing number of tasks efficiently, lightweight fine-tuning approaches such as LoRA [48] or sparse transformers such as Longformer [49] could be employed to enable faster and more scalable fine-tuning.

Although we primarily focus on the x86 architecture due to space constraints, ProTST is readily applicable to other instruction sets, such as ARM and MIPS, particularly with datasets like [41] 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 straightforward manual extraction. Future work will explore the application of ProTST to these architectures.

ProTST leverages parameter-based knowledge transfer. Alternative methods such as feature-based and unified-loss

learning offer distinct approaches to knowledge transfer. In feature-based transfer learning, each task generates feature embeddings as supplementary input for subsequent tasks within the hierarchical structure. This facilitates the propagation of knowledge from earlier to later tasks through these learned representations. Unified-loss learning, on the other hand, treats all task-specific loss functions as a single, unified loss, thereby supervising all tasks simultaneously. This approach allows knowledge transfer to benefit from the backpropagation of gradients from later tasks to earlier ones, potentially leading to further refinements in the learned representations. However, both require the design of a specialized model architecture and a training procedure. We are actively investigating the implementation and evaluation of both feature-based and unified-loss within the PROTST framework.

VII. CONCLUSION

We introduce PROTST, a Progressive Teacher-Student Binary Analysis framework specifically designed to enhance both the accuracy and efficiency 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 underscore the efficacy of PROTST in a broad range of binary analysis tasks. The results of intensive testing reveal that our progressive teacher-student framework significantly exceeds existing methods regarding learning efficiency and task performance. We believe that our method opens up new avenues for research and offers a promising starting point for future work.

ACKNOWLEDGMENT

We thank our shepherd and the 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.

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, "Symlm: 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] 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.

[17] 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.

[18] 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.

[19] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *USENIX Security*, 2015.

[20] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *USENIX Security*, 2017.

[21] Y. Otsubo, A. Otsuka, M. Mimura, and T. Sakaki, "o-glasses: Visualizing x86 code from binary using a 1d-cnn," *IEEE Access*, 2020.

[22] 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.

[23] 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.

[24] 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 *ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2017.

[25] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *ACM/IEEE Conference on Automated Software Engineering (ASE)*, 2018.

[26] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *Network and Distributed System Security (NDSS)*, 2023.

[27] S. Yang, C. Dong, Y. Xiao, Y. Cheng, Z. Shi, Z. Li, and L. Sun, "Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2023.

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

[29] W. Zhu, Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang, "Callee: Recovering call graphs for binaries with transfer and contrastive learning," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[30] T. Ye, L. Wu, T. Ma, X. Zhang, Y. Du, P. Liu, S. Ji, and W. Wang, "Cp-bcs: Binary code summarization guided by control flow graph and pseudo code," *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[31] J. Xiong, G. Chen, K. Chen, H. Gao, S. Cheng, and W. Zhang, "Hext5: Unified pre-training for stripped binary code information inference," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[32] J. Schmidhuber, S. Hochreiter *et al.*, "Long short-term memory," *Neural Comput*, 1997.

[33] 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.

[34] 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.

[35] 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.

[36] D. Gibert, "Machine learning for windows malware detection and classification: Methods, challenges and ongoing research," *arXiv preprint arXiv:2404.18541*, 2024.

[37] 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.

[38] "Gnu binutils," https://www.gnu.org/software/binutils/, 2023, [Online; accessed 25-Jun-2024].

[39] "Spec cpu2017 benchmark suite," https://www.spec.org/cpu2017/, 2017, [Online; accessed 25-Jun-2024].

[40] "Spec cpu2006 benchmark suite," https://www.spec.org/cpu2006/, 2006, [Online; accessed 25-Jun-2024].

[41] 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.

[42] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," in *arXiv 1802.10135*, 2018.

[43] 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.

[44] 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.

[45] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning (ICML)*, 2014.

[46] 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.

[47] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 41–48.

[48] 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," *arXiv preprint arXiv:2106.09685*, 2021.

[49] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv:2004.05150*, 2020.

APPENDIX A
BINARY TASK HEAD DETAILS

PROTST contains different heads for seven different binary tasks integrated into its model architecture.

Instruction Boundary Recovery Head. Instruction boundary recovery (❷) aims to identify the starting points of the instructions and whether subsequent bytes belong to the same instruction sequence. Give an input byte sequence x of length n, the corresponding ground truth y^{IB} is another sequence of labels of the same length n:

$$y^{IB} = \{y_1^{IB}, y_2^{IB}, ..., y_n^{IB}\} \tag{11}$$

where each element y_i^{IB} in the sequence belongs to the set $\{SI, MI\}$. Here, "SI" denotes the "Start of Instruction" and "MI" denotes "Middle of Instruction". Each byte x_i in the input sequence is assigned a corresponding label y_i^{IB} in the ground-truth sequence. To train the model effectively for this token-level classification task, a cross-entropy loss function is employed. This function measures the difference between the predicted probabilities for each byte's class (SI or MI) and the actual ground truth labels. Mathematically, it can be represented as:

$$\mathcal{L}_{IB} = -\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}^{IB} \log P(y_{i,c}^{IB} \mid x_i) \tag{12}$$

where C is the number of classes (2 in this task, SI and MI) and $y_{i,c}^{IB}$ is a binary indicator (0 or 1) that is 1 if the true class of the i-th byte is c. In simpler terms, this objective compares the true class labels for each byte x_i against the predicted probabilities across all possible classes C (SI and MI).

Function Boundary Recovery Head. Function boundary recovery (❸) aims to identify the starting point, the continuation within the function, and the end point of functions within binary files. Given an input byte sequence x of length n, the corresponding ground truth label y^{FB} is another sequence of labels of the same length n. This can be mathematically expressed as:

$$y^{FB} = \{y_1^{FB}, y_2^{FB}, ..., y_n^{FB}\} \tag{13}$$

where each element y_i^{FB} in sequence belongs to the set $\{SF, MF, EF\}$. Here, "SF" denotes the "Start of Function","MF" denotes the "Middle of Function", and "EF" denotes the "End of Function". Each byte x_i in the input sequence is assigned a corresponding label y_i^{FB} in the ground-truth sequence. To train the model effectively for this token-level classification task, a cross-entropy loss function is employed. This function measures the difference between the predicted probabilities for each byte's class (SF, MF or EF) and the actual ground truth labels. Mathematically, it can be represented as:

$$\mathcal{L}_{FB} = -\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}^{FB} \log P(y_{i,c}^{FB} \mid x_i) \tag{14}$$

where C is the number of classes (3 in this task, SF, MF and EF) and $y_{i,c}^{IB}$ is a binary indicator (0 or 1) that is 1 if the true class of the i-th byte is c. In simpler terms, this objective compares the true class labels for each byte x_i against the predicted probabilities across all possible classes C (SF, MF and EF).

Function Signature Prediction Head. Function signature prediction (❹) aims to identify the number of arguments a function takes and its return data type. Given a binary function represented by its byte sequence x, the corresponding ground truth y^{FS} that defines the function's signature. This can be mathematically expressed as:

$$y^{FS} = \{y_n^{FS}, y_t^{FS}\} \tag{15}$$

where y_n^{FS} represents the number of arguments the function takes. It can take values from the set {1, 2, 3, 4, 5, "others"}. The "others" category encompasses functions with more than 5 arguments. y_t^{FS} represents the return data type of the function. It can take values from the set {int, char, void, double, bool, "others"}. The "others" category encompasses the less common data types. To train the model effectively for this sequence-level classification task, a cross-entropy loss function is used. This function measures the difference between the predicted probabilities for each aspect of the signature (number of arguments and return type) and the actual ground truth labels. Mathematically, it can be represented as:

$$\mathcal{L}_{FS} = -\Bigg[\sum_{c=1}^{C_n} y_{n,c}^{FS} \log P(y_{n,c}^{FS} \mid x) + \sum_{d=1}^{C_t} y_{t,d}^{FS} \log P(y_{t,d}^{FS} \mid x)\Bigg] \tag{16}$$

where C_n is the number of classes for the number of arguments (e.g., 1, 2, 3, 4, 5, "others"), C_t is the number of classes for the return data type (e.g., int, char, void, double, bool, "others"). $y_{n,c}^{FS}$ is a binary indicator (0 or 1) that is 1 if the true class of the number of arguments is c. $y_{t,d}^{FS}$ is a binary indicator (0 or 1) that is 1 if the true class of the return data type is d. This cross-entropy loss function combines the losses for both predictions into a single loss function by summing the individual cross-entropy losses for each attribute.

Function Similarity Detection Head. Function similarity detection (❺) aims to assess the degree of similarity between two code snippets represented in binary code. Given a pair of functions x_1 and x_2, the backbone model processes each function separately, generating embeddings E^{x_1} and E^{x_2}. These embeddings capture the essential characteristics of the respective functions. The model then determines the similarity between the functions. The ground truth label y takes on a value of either 1 or -1 (1 indicates similar functions and -1 indicates dissimilar functions). To calculate the function embedding, the model first averages the individual embeddings generated by the backbone model for each function E_i^x. This averaged embedding is then fed into a 2-layer Multi-Layer Perceptron (MLP) network. The MLP network further processes the averaged embedding to produce a final output that reflects the predicted similarity between the functions. The final embedding of function can be mathematically expressed as:

$$\mathbf{F}(x) = \text{MLP}\left(\frac{1}{N}\sum_{i=1}^{N} \mathbf{E}_i^x\right) \tag{17}$$

We use cosine embedding loss to train this task. Given two function embeddings, the loss can be formulated as:

$$\mathcal{L}_{CE} = \begin{cases} 1 - \cos(\mathbf{F}(x_1), \mathbf{F}(x_2)) & \text{if } y = 1 \\ \max(0, \cos(\mathbf{F}(x_1), \mathbf{F}(x_2)) - m) & \text{if } y = -1 \end{cases}$$

where $\cos(\mathbf{F}(x_1), \mathbf{F}(x_2))$ denotes the cosine similarity between the embeddings $\mathbf{F}(x_1)$ and $\mathbf{F}(x_2)$. m is a margin that helps to distinguish dissimilar pairs, typically $0 \leq m \leq 1$. This cosine embedding loss encourages the model to maximize the similarity for similar functions and minimize it for dissimilar ones.

Function Name Prediction Head. Function name prediction (❻) aims at predicting the names assigned to functions within binary code. Given a binary function represented by its byte sequence x, the corresponding ground truth y^{FN} is a sequence of words that represents the function name. This can be mathematically expressed as:

$$y^{FN} = \{w_1^{FN}, w_2^{FN}, ..., w_n^{FN} \mid w_i^{FN} \in V\}, \tag{18}$$

where w_i^{FN} represents an individual word within the sequence of predicted function names. V represents the complete vocabulary of possible function names.

This vocabulary consists primarily of English words but may also include: (1) Developer-chosen terms such as abbreviations, data types (e.g., int, float), (2) numbers, and (3) misspellings. To address challenges such as morphological variations (word form differences) and frequent occurrences of out-of-vocabulary (OOV) words in function names, we follow the pre-processing strategy [4], which involves splitting, segmenting, and lemmatizing (converting words to their base form) function names. Additionally, we adjust thresholds to filter out function names containing tokens that appear either too frequently or infrequently in the training data. This helps ensure a more balanced and fair learning process for the model. The task is framed as a multi-class, multi-label classification problem. We employ the BCElogit loss to solve this task. Mathematically, this loss function can be formulated as:

$$\mathcal{L}_{FN} = -\sum_{i=1}^{n}\sum_{j=1}^{|V|}\Big[y_{i,j}^{FN} \log \sigma(P(w_{i,j}^{FN} \mid x)) + (1 - y_{i,j}^{FN}) \log(1 - \sigma(P(w_{i,j}^{FN} \mid x)))\Big] \tag{19}$$

where n is the length of the function name sequence, $|V|$ is the size of the vocabulary V. $y_{i,j}^{FN}$ is a binary indicator (0 or 1) that is 1 if the i-th word in the function name is the j-th word in the vocabulary V. σ represents the sigmoid function, which converts logits into probabilities. This loss function measures the discrepancy between the predicted probabilities and the actual labels for each word in the function name.

Compiler Provenance Head. Compiler provenance (❼) aims to identify the compiler and optimization level used to generate a binary file. Given a binary file represented by its byte sequence x, the model predicts a pair of labels y^{CP}. This pair identifies the compiler and the optimization level used for compilation. This can be expressed as:

$$y^{CP} = \{y_c^{CP}, y_o^{CP}\} \tag{20}$$

where y_c^{CP} represents the compiler used. For instance, it can take values from the set {clang, gcc}, indicating either the "clang" or "gcc" compiler. y_o^{CP} represents the optimization level. It takes values from the set $\{O_0, O_1, O_2, O_3, O_s, O_{fast}\}$, signifying different optimization levels offered by the

compilers. To train the model effectively for this sequence-level classification task, a cross-entropy loss function is used. This function measures the difference between the predicted probabilities for each aspect of the signature (compiler and its optimization) and the actual ground truth labels. Mathematically, it can be represented as:

$$\mathcal{L}_{CP} = -\Bigg[\sum_{c=1}^{C_c} y_{c,c}^{CP} \log P(y_{c,c}^{CP} \mid x) + \sum_{o=1}^{C_o} y_{o,o}^{CP} \log P(y_{o,o}^{CP} \mid x)\Bigg] \tag{21}$$

where C_c is the number of compiler classes (e.g., clang, gcc), C_o is the number of optimization level classes (e.g., O_0, O_1, O_2, O_3, O_s, O_{fast}). $y_{c,c}^{CP}$ is a binary indicator (0 or 1) that is 1 if the true class for the compiler is c. $y_{o,o}^{CP}$ is a binary indicator (0 or 1) that is 1 if the true class for the optimization level is o. This loss function combines the errors from the predicted probabilities of both the compiler and the optimization level into a single objective.

Malware Classification Head. Malware classification (❽) aims to categorize a binary file into a specific type of malware family based on its byte sequence. Given a malware file represented by its byte sequence x, the model predicts a label y^{MC}. This label corresponds to a specific malware family from a predefined list in the BIG2015 dataset [42]. The list includes malware families like Ramnit, Lollipop, Kelihosver3, and others. To train the model effectively for this sequence-level classification task, a cross-entropy loss function is used, which can be formulated as

$$\mathcal{L}_{MC} = -\sum_{k=1}^{K} y_k^{MC} \log P(y_k^{MC} \mid x) \tag{22}$$

where K is the number of malware family classes (9 in this task, BIG2015 [42] contains 9 malware families). y_k^{MC} is a binary indicator (0 or 1) that is 1 if the correct class for the malware family is k. This loss function evaluates the discrepancy between the predicted probabilities for each malware family and the actual ground truth labels, guiding the model to improve its classification accuracy.

APPENDIX B
DATASET DETAILS

We present the details of the datasets used to evaluate PROTST below.

Binutils. This dataset [38] is generated by compiling the GNU Binutils package with its default settings. It serves as the training data for the MLM pre-training stage.

SPEC CPU. This dataset includes binaries from various benchmarks (SPEC CPU 2017 [39] and 2006 [40]), compiled with different configurations. SPEC CPU 2017 includes 588 binaries, while SPEC CPU 2006 has 333. They are used for instruction and function boundary recovery.

BAP. This dataset [12] contains 2,200 binaries from open-source programs across various platforms (Windows, Linux) and architectures (x86, x64). It's used for function boundary recovery.

BIG2015. This dataset [42] consists of 10,868 malware samples categorized into nine families. It's used for malware classification.

Binkit. This dataset [41] features 243,128 binaries with 75,230,573 binary functions derived from 51 distinct software packages, compiled using a diverse array of options across compilers and architectures. It's used for tasks like compiler provenance and function signature prediction.

SymLM. This dataset [4] includes 16,027 binaries and 1,431,169 functions derived from 27 open-source projects, compiled across multiple architectures and optimization levels using gcc-7.5. It's used for function name prediction.

Binarycorp-3M. This dataset [8] encompasses approximately 3.6 million functions extracted from 10,265 binary programs compiled using gcc and g++ based on ArchLinux packages. It's used for function similarity detection.