BLens: Contrastive Captioning of Binary Functions using Ensemble Embedding

Tristan Benoit*†¶, Yunru Wang*†‡, Moritz Dannehl†‡, Johannes Kinder†‡

† *Ludwig-Maximilians-Universität München, Germany*
‡ *Munich Center for Machine Learning, Germany*
¶ *Bundeswehr University Munich, Germany*

Abstract

Function names can greatly aid human reverse engineers, which has spurred the development of machine learning-based approaches to predicting function names in stripped binaries. Much current work in this area now uses transformers, applying a metaphor of machine translation from code to function names. Still, function naming models face challenges in generalizing to projects unrelated to the training set. In this paper, we take a completely new approach by transferring advances in automated image captioning to the domain of binary reverse engineering, such that different parts of a binary function can be associated with parts of its name. We propose BLens, which combines multiple binary function embeddings into a new ensemble representation, aligns it with the name representation latent space via a contrastive learning approach, and generates function names with a transformer architecture tailored for function names. Our experiments demonstrate that BLens significantly outperforms the state of the art. In the usual setting of splitting per binary, we achieve an F_1 score of 0.79 compared to 0.70. In the cross-project setting, which emphasizes generalizability, we achieve an F_1 score of 0.46 compared to 0.29. Finally, in an experimental setting reducing shared components across projects, we achieve an F_1 score of 0.32 compared to 0.19.

1 Introduction

Reverse engineers analyze programs available only in binary form in security audits, for vulnerability discovery, or during forensic analysis of malware [2, 14, 67]. Disassemblers such as IDA Pro or Ghidra provide assistance by discovering functions within the binary and resolving data references and control flow. Programs being analyzed have usually been stripped of most human-readable information, such as meaningful function names. Because function names help to effectively navigate the code and identify points of interest, human reverse engineers are known to manually label functions in the disassembly with names that capture their semantics [52, 74].

Machine learning promises to help automate tasks that require a human level of understanding, especially with imprecise concepts such as developer-assigned names. Learned embeddings of binary code aim to capture the semantics of assembly code in a compact vector representation [17, 44, 51, 53, 75, 76, 85]. Binary code embeddings can be used directly for binary code similarity detection or as part of more complex architectures and tasks.

Pioneering work on naming binary functions aims to learn and predict complete function names [24, 28, 56], an approach which is by design limited to a coarse granularity of semantics and mostly recovers function names frequently seen in the training data [58]. Splitting function names into sets of labels or tokens addresses these limitations [15, 58]. However, a multi-label setting ignores the ordering of words inside a function name, and ordering during post-processing misses important nuances in function names.

Recent transformer-based approaches use an encoder-decoder architecture [72] to effectively treat function naming as a translation problem from binary code to function names, which are seen as sentences of tokens [15, 35, 37, 83]. A key design choice in this line of work lies in the binary code representation available to the encoder. For instance, AsmDescriptor [37] works on assembly code, while HexT5 [83] leverages a decompiler to work directly on pseudocode.

In this paper, we argue that translation is *not* the right metaphor in function name prediction but that the task is closer to generating meaningful captions for images [36]. In particular, our intuition is that *binary code and function names should be treated as two modalities for the same concept*.

By integrating ideas from the multimodal machine learning field [6], we aim to obtain a solid relation between parts of a function and words. For instance, Contrastive Language-Image Pre-Training (CLIP) [63] consists of breaking down images into smaller patches and then associating them with corresponding human-readable text during pre-training. Vision transformers [19] process image patches in a way that

*Both authors contributed equally to this work.

Published in 34th USENIX Security Symposium, Seattle, WA, USA, August 13 – 15, 2025

captures spatial relationships between them. Just as image captioning requires understanding different visual components (e.g., colors and shapes) and their visual relationships to generate a coherent description, function name prediction involves associating parts of binary code with function names in a way that captures interrelationships in the binary code. These relationships reside at the level of control and data flows, as well as within operations, function calls, and resources, such as strings. Still, there are differences between images and functions that one should consider. First, it is natural to obtain image patches by cutting images, but a function can be characterized by various structures, from sequences to graphs. Second, currently available function datasets comprise millions of functions at most, while image datasets comprise billions of images [91]. Third, while results for function naming published so far are promising, they still do not generalize well across separate projects, as we explain below.

Distribution shifts. Most work in learning-based binary analysis adopts the **cross-binary setting**, where functions in the training, validation, and test sets are drawn from distinct binaries. Xiong et al. [83] argue that the **cross-project setting**, where an entire project is assigned to a single set, better reflects real-world use cases, as reverse engineers typically encounter binaries from unknown projects. This setting is difficult for learning-based methods as it demands better generalization capability from models. Binary code within the same project tends to have similar distributions due to shared components, coding practices, and compilation configurations, which may lead to overfitting. Consequently, accuracy drops significantly in the cross-project setting, as the training and test sets exhibit substantial distribution differences, referred to as **distribution shifts**. While different projects may still share individual source code components or snippets, this is expected even in real-world scenarios. However, to evaluate solely generalization capabilities, we additionally introduce a strict evaluation setting (§6.4), in which we aggressively minimize the impact of shared components across projects.

Challenges. We identify the following challenges to function name prediction on stripped binary code: (C1) Function names often encapsulate semantics sensitive to the order of words (e.g., `int_to_float`). It is thus important to capture word order through the binary code structure. (C2) Both precision and recall are crucial for providing useful function names. A good precision guarantees that suggested names are relevant and not misleading, while a good recall ensures the model proposes names for most functions. Achieving a satisfactory amount of both is challenging. (C3) Ensuring robust generalization is critical to the more challenging cross-project setting. Yet existing approaches are seldom evaluated in this realistic setting [15, 24, 28, 35, 56, 58].

Our approach. We introduce BLens (**B**inary **Lens**), which aligns the modality of binary code—represented as function patches—with human-readable text through the contrastive captioning (CoCa) [88] multi-task. To obtain these patches, BLens leverages state-of-the-art embeddings: CLAP [75] for a cross-modal function embedding, PALMTREE [44] to represent sequences of basic blocks, and DEXTER [58] for a context-aware function embedding. Because multi-task models are robust [8] but suffer from conflicts among training objectives [65], we fine-tune BLens's decoder strictly for captioning. In order to handle rather small datasets and short function names, the decoder learns through a Masked Language Modeling (MLM) task [16] tailored to function names, and to maintain precision, it employs a confidence threshold. In particular, we make the following contributions:

- We present BLens, a new approach to function name prediction inspired by multimodal machine learning (§2). Our architecture integrates multiple existing binary code representations thanks to the COMBO (**CO**ntrastive **M**ulti-modal **B**inary embedding **O**ptimizer) pre-training phase (§4) to achieve better generalization.
- We introduce LORD (**L**ikelihood **O**rdered **R**egressive **D**ecoder) (§5), a decoder and inference scheme that employs a hard MLM task for a more profound fine-tuning process along with a flexible autoregression. LORD maintains consistently high precision even in the challenging cross-project setting.
- BLens achieves substantial improvements over the state of the art, particularly on the grammatical structure. In an evaluation in the cross-binary setting (§6.2), we observe gains of 12% F_1, 32.6% RougeL, 79.1% Bleu, and 11.1% VarCLR scores. In the cross-project setting (§6.3), we observe gains of 42.2% F_1, 71.7% RougeL, 188% Bleu, and 12.1% VarCLR scores. In the strict setting, which removes shared components (§6.4), BLens provides a gain of 53.3% in terms of F_1 score. An ablation study (§6.5) validates our contributions by demonstrating a 55% increase in F_1 score from the COMBO pre-training and a 56.2% boost in precision from the LORD decoder.

Additionally, a case study of predictions in the cross-project setting (§7) illustrates BLens's capacity to generate relevant function names that may differ from the ground truth, thereby addressing the lower F_1 score in this hard setting.

BLens, along with all experimental data, is available as open source [7].

2 Multimodal Machine Learning

This section introduces two approaches from multimodal machine learning for image captioning, CoCa [88] and GIT [77], which form the foundation of BLens. As both are built around the transformer [72] as their core component, we begin by recalling its technical details.

Transformer. The transformer [72] has recently found widespread application in binary analysis [3, 35, 44, 59–61,

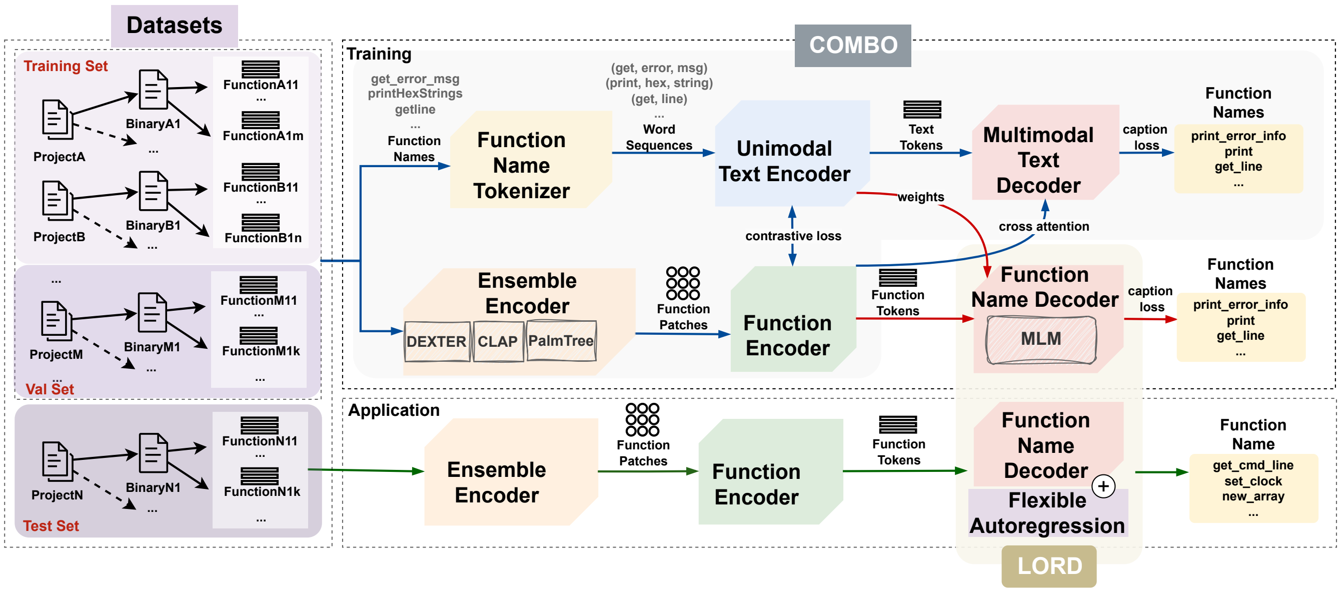

Figure 1: Overview of BLens. Inspired by CoCa [88] and GIT [77], BLens employs a two-stage process: pre-training (blue arrows) and fine-tuning (red arrows). In pre-training, we initialize function tokens using an ensemble encoder and pre-train the function encoder with a contrastive captioning task. During fine-tuning, the pre-trained encoder, combined with a function name decoder trained with a new MLM task, generates function names. In inference (green arrows), the model takes functions as input and outputs their corresponding names with the flexible autoregression.

76, 89, 94, 97], demonstrating its effectiveness in learning binary representations. It consists of an encoder and a decoder, each with several stacked blocks. The main components of encoder (and decoder) blocks are multi-head attention, residual connections [29], and layer normalization [4]. Multiple attention heads perform self-attention independently, allowing the model to capture a wider range of dependencies by attending to different aspects of the input sequence simultaneously. Residual connections, implemented by adding the layer input directly to its output, help maintain gradient flow and prevent degradation in deep networks. Layer normalization, which operates by normalizing the summed inputs across each neuron before applying activation, ensures consistent activation values and stabilizes training.

CoCa. CoCa [88] includes an image encoder, a unimodal text decoder, and a multimodal text decoder. The image encoder processes image patches, while the uni- and multimodal decoders form the two halves of a transformer decoder. The unimodal decoder encodes text using masked self-attention, and the multimodal decoder combines the outputs of the unimodal decoder and the image encoder with cross-attention to create image-text representations. The objective of CoCa combines contrastive and captioning losses. The contrastive loss aligns text and image tokens in the same latent space, while the captioning loss is a generative loss derived from predicting captions.

GIT. Compared to CoCa, GIT [77] is a simpler generative model for image/video captioning, comprising only one image encoder and a text decoder. It concatenates image tokens with text tokens as input to the transformer module and predicts the associated description for the input image. Notably, GIT employs an existing encoder pre-trained with a contrastive pre-training task [90] as the image encoder, similar to how we pre-train a function encoder based on the contrastive captioning (CoCa) task.

3 Overview

Now, we first define the problem and then introduce the training and application process of our model.

Problem Definition. Function name prediction automatically recovers function names from stripped binary code. Formally, the process of recovering function names from functions is defined as

$$Name_i = G(F_i^{p,b})$$

where $F_i^{p,b}$ represents the i-th function extracted from binary b in project p, G denotes our model, $Name_i$ is the ground truth name for $F_i^{p,b}$. The input of the model is a function, encompassing related code, data flow, control flow, and other features. The output of the model is a sequence of words $\hat{Name_i} \in Vocab^n$ where $Vocab$ is the set of words from the vocabulary and n is the sequence length. The effectiveness of the model is judged based on how closely its output aligns with the ground truth.

Terminology. We draw inspiration from standard terms used in image captioning tasks to denote intermediate outputs. Similar to how image patches are inputs to the image encoder, we refer to the inputs of the function encoder as function patches. Moreover, the outputs of encoders are termed tokens. Finally, each token is a vector of fixed dimensionality d.

Training stages. Training consists of a pre-training phase followed by a fine-tuning phase. During pre-training, COMBO learns robust function representations (with blue arrows in Figure 1) with a contrastive captioning task. During fine-tuning, the new decoder LORD is added on top of the pre-trained function encoder to generate function names (with red arrows in Figure 1). Throughout the entire training process, all weights are updated.

COMBO. Firstly, given $F_i^{p,b}$ as input, the ensemble encoder embeds processed function features with three state-of-the-art embeddings and then generates function patches. Secondly, the tokenizer turns the ground truth name into the sequence of words $Name_i$ that the unimodal text encoder converts into text tokens. Simultaneously, the function encoder transforms the function patches into function tokens. There, a contrastive task relates the outputs of both encoders. Thirdly, a multimodal text decoder integrated on top of both encoders generates multimodal text tokens. Lastly, a captioning task relates $Name_i$ to these tokens. Details are provided in §4.

Previous contrastive pre-trainings on functions [75, 95] do not combine contrastive and captioning losses, missing out on the advantages of a multi-task approach [8]. Additionally, they require robust summaries generated by LLMs as anchors, which are challenging to obtain [26]. In contrast, we directly learn text tokens as anchors through the unimodal text encoder.

LORD. Training on multiple tasks introduces conflicts [65]. Therefore, we introduce a fine-tuning stage with a function name decoder, LORD, that is strictly focused on captioning through a novel Masked Language Modeling (MLM) task. This task replaces the standard teacher forcing in transformers with predicting randomly selected words from the remaining context. Additionally, during application, the task allows LORD to employ a flexible autoregression process, which picks probable words one step at a time. The flexible autoregression stops when the confidence scores of the proposed words fall below a fine-tuned threshold. The details are shown in §5.

Application. In Figure 1, the dashed box and green arrows at the bottom illustrate the actual application of the model. Given $F_i^{p,b}$ as input, the ensemble encoder begins by embedding processed function features with three state-of-the-art embeddings. Then, it generates function patches from the initial embeddings. Moreover, these patches are then transformed into function tokens by the function encoder. Finally, following the flexible autoregression process, the function name decoder generates text tokens one at a time, which are then converted into a word sequence, resulting in $\hat{Name}_i$.

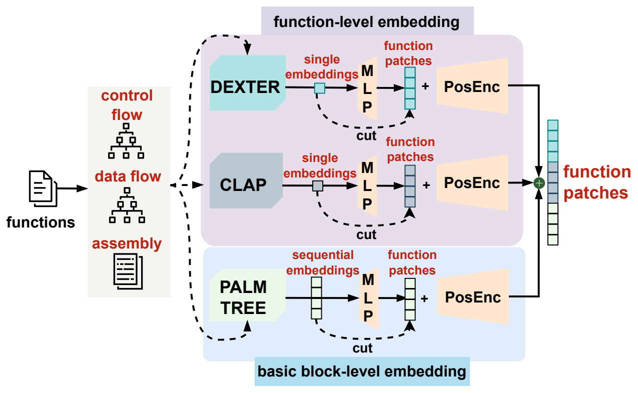

Figure 2: Overview of the ensemble encoder.

By using the idea of image captioning, we capture the word order through the binary code structure, addressing challenge **C1**. With COMBO, we aim to address distribution shifts, mentioned in challenge **C3**, by associating function parts with similar semantics to corresponding words in names. The LORD task and autoregressive process significantly boost precision (see §6.5.3), which is key to challenge **C2**.

4 COMBO

We now present COMBO in detail. With stripped binaries as input, the ensemble encoder outputs function patches using existing SotA approaches (§4.1). By utilizing these existing models instead of building a custom binary representation model, we can leverage the diverse binary features extracted by different models, thereby enriching the representation and enhancing robustness. Next, we refine the function patches with a contrastive captioning task (§4.2), which links function tokens to corresponding text tokens. The robust semantic understanding of COMBO mitigates distribution shifts [84].

4.1 Ensemble Encoding

To better represent functions, BLens leverages knowledge from existing binary representation models instead of training new ones from scratch. Different models use distinct binary pre-processing techniques and extract unique features from functions. Thus, selecting suitable combinations is crucial to avoid redundancy and ensure complementary features.

We categorized state-of-the-art binary representation models by the granularity of their embeddings. Some approaches generate embeddings at the basic block level, while others produce a single embedding for an entire function. Based on the differences in embedding granularity, extracted binary features, accessibility, and performance, we selected PALMTREE [44], CLAP [75], and DEXTER [58]. Although we focused on these three models, our module can be applied to various embedding approaches.

For basic block embeddings, we select PALMTREE [44], a pre-trained BERT-based [16] assembly language model. It generates instruction embeddings through self-supervised learning incorporating both data-flow and control-flow information, allowing for straightforward extraction of basic block embeddings. For function-level embeddings, we select CLAP and DEXTER. CLAP [75] is the best-performing approach we have identified in our experiments, effectively aligning function code with descriptive text. DEXTER [58] employs static analysis to extract both quantitative (e.g., number of transitively reachable functions) and categorical features (e.g., MinHash hashes of assembly opcodes) from functions and uses MLP layers to integrate these features and produce function embeddings. Compared to most methods that use learning-based tasks to capture binary features, DEXTER employs complex static analysis. It is also one of the best-performing binary embeddings for function name prediction. The three models have distinct focuses: PALMTREE emphasizes control flow and data flow dependencies among assembly instructions, CLAP incorporates source code information, and DEXTER focuses on statistical features.

Our attention mechanism requires many input patches to discern relationships effectively. While images, videos, and speeches can easily be cut into a sequence [10, 19, 42], this is not the case for function-level embeddings. A projection [22, 43, 73] either fails to produce enough patches or requires so many parameters that it dilutes the effectiveness of the attention mechanism. To effectively combine these embeddings, we use neural networks structured as in Figure 2, where PosEnc stands for the positional encoding. For function-level embeddings, we apply the concept of image patches, cutting embeddings into smaller embeddings and projecting them to function patches by an MLP layer. For basic block-level embeddings, an MLP layer projects them to patches. Learnable positional encodings are then added to each patch. Notably, after a function-level embedding has been turned into patches, the positional encoding of each patch indicates which part of the original embedding it contains. Finally, all patches are concatenated for joint training on the contrastive captioning task during pre-training.

4.2 Contrastive Captioning

Function name captioning can also serve as a pre-training task, as in CoCa [88]. Contrastive captioning is the backbone of our robust modality translation from binary code to text. In this section, we outline the detailed process of the contrastive captioning task.

Inspired by CoCa, we use a unimodal text encoder to encode words into text tokens and a function encoder to generate function tokens. By applying contrastive loss, we align the text tokens with function tokens. Additionally, we add a multimodal text decoder on top of the dual encoders to perform the captioning task, further enhancing the ability of the model to generate accurate and relevant function names.

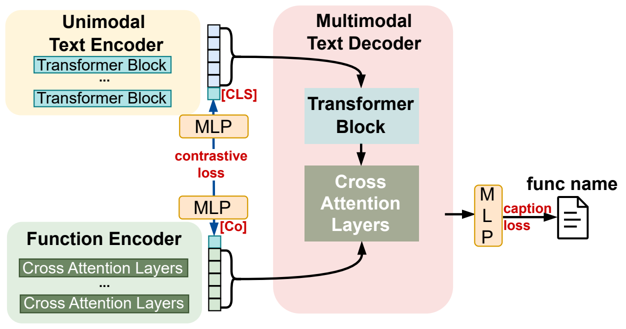

Figure 3: Overview of the contrastive captioning task.

The structures of the encoders and the decoder are shown in Figure 3. In the dual-encoder setup, given a function and its name, the name is first tokenized into words and then embedded into $n+1$ initial text tokens, where n is the maximum number of words in a name (padded if fewer), and the +1 accounts for the addition of a [CLS] token at the end of each token sequence. These initial text tokens serve as the input for the unimodal text encoder, which produces unimodal text tokens through transformer blocks.

For the function encoder, the function is first transformed into k_1 function patches of dimension d using the ensemble encoder. Remember that d is also the dimension of each token. Function patches are then passed as input to the function encoder, which outputs k_2 function tokens through cross-attention with k_2 learnable queries. Among these function tokens, one is reserved for the contrastive task, referred to as the contrastive token, while others are reserved for the captioning task.

The contrastive task operates over a batch of B functions and names; it consists of aligning [CLS] text tokens CLS with corresponding contrastive tokens Co:

$$\mathcal{L}_{\text{Cross-Entropy}}(x,y) = -\frac{1}{B}\sum_{i=1}^{B}\log\frac{\exp(x_i^{\mathsf{T}} y_i/\sigma)}{\sum_{j=1}^{B}\exp(x_i^{\mathsf{T}} y_j/\sigma)}$$

$$\mathcal{L}_{\text{Contrastive}} = \mathcal{L}_{\text{Cross-Entropy}}(CLS, Co) + \mathcal{L}_{\text{Cross-Entropy}}(Co, CLS)$$

Here, i and j are indices for functions and names in the i-th and j-th pairs, while σ is the temperature used to scale the logits. $L_{Contrastive}$ is the sum of function-to-name cross-entropy and name-to-function cross-entropy. This approach minimizes the distance between x_i and y_i while maximizing the distance between x_i and y_j. Such alignment ensures that the function and text tokens in each pair remain in the same latent space.

The captioning loss uses function tokens, excluding the contrastive one, and unimodal text tokens, excluding [CLS].

As shown in Figure 3, we first connect the multimodal text decoder to the unimodal text encoder, and then link the multimodal text decoder to the function encoder via cross-attention layers. As a result, the multimodal text decoder can leverage information from both the unimodal text encoder and the function encoder. Thus, during backpropagation, the weights of both encoders are updated accordingly. The output of the multimodal decoder is function names, as word sequences, and the loss function is the standard cross-entropy loss for generation tasks.

Given ground truth words *Name* and multimodal text tokens *MMTTokens*, the caption loss is as follows:

$$\mathcal{L}_{\text{Caption}} = -\sum_{t=1}^{n} \log \mathcal{P}_\theta(Name_t \mid Name_{<t}, MMTTokens)$$

Here, $Name_t$ is the ground truth word at time step t and $Name_{<t}$ denotes previous words up to time step $t-1$. $\mathcal{P}_\theta(Name_t \mid Name_{<t}, MMTTokens)$ is the predicted probability of $Name_t$ given $Name_{<t}$ and *MMTTokens*.

The full loss function of the training process is as follows:

$$\mathcal{L}_{\text{Full}} = \mathcal{L}_{\text{Contrastive}} + \mathcal{L}_{\text{Caption}}$$

After COMBO, the function encoder can project function tokens into a latent space closely aligned with that generated by the unimodal text encoder. This alignment opens possibilities for zero-shot function-text retrieval [75].

The multimodal text decoder can already generate imprecise function names. However, pre-trained models perform best when fine-tuned only on their downstream task, which is why CoCa is fine-tuned on only the captioning task [88]. Consequently, we fine-tune the pre-trained model on function name prediction inspired by the encoder-decoder structure in GIT [77]. Along the way, we propose a new decoder, LORD, that is more appropriate for function name prediction.

5 LORD

An important challenge in function name prediction is achieving an optimal balance between recall and precision. Specifically, it is crucial to prevent overwhelming a reverse engineer with excessive false positives, which are time-consuming to verify. Preliminary work [87] indicates that traditional transformer approaches tend to be overconfident, prioritizing recall excessively. This issue is common to modern neural networks, especially in the face of distribution shifts [40, 80]. This bias is acceptable in problems such as translation and image captioning, where errors are easily noticeable and distribution shifts are small. However, in binary analysis, a transformer should maintain precision by being more cautious.

We propose LORD, a new decoder and inference scheme parameterized by a confidence threshold. The embedding layer of this decoder is initialized with weights from the unimodal text encoder. To deepen semantic understanding, we introduce a new Masked Language Modeling (MLM) task (detailed in §5.1), inspired by BERT [16], to replace left-to-right teacher forcing. During inference, we use a more precise autoregressive process referred to as flexible autoregression. It selects the word with the highest confidence score at each step until the score falls below the threshold. Details are provided in §5.2.

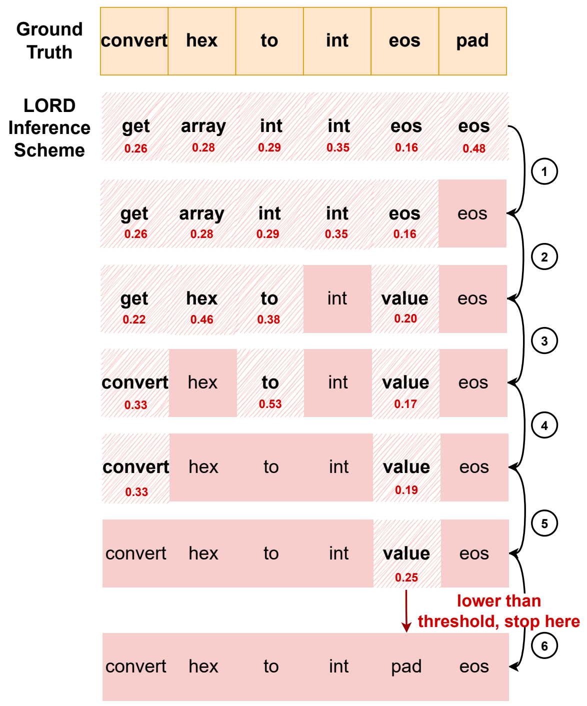

Figure 4: A toy example of the flexible autoregressive process.

5.1 Masked Language Modeling

During training, a traditional generative transformer learns through teacher forcing [38]. Each word is predicted with access to the preceding words as context. During inference, autoregression predicts words sequentially from left to right until the [EOS] word is reached. This approach is valid because, with enough transformer blocks, autoregression can express any sequence distribution. However, teacher forcing simplifies the prediction of later words compared to the first word and can fail planning tasks [5].

We propose a new MLM task to replace teacher forcing. We train the decoder to predict masked words given unmasked ones. This allows for more possibilities of masking and helps to mitigate the issue of short function names. Additionally, this lets the decoder learn deeper insights into the relationship between function names and binary code.

In line with the traditional transformer, where the context (previous words) never includes [EOS] and [PAD] words, we consistently mask these words. However, traditional MLM tasks randomly mask 15% of words within a long text corpus [81], yet function names are typically short, often consisting of three to five words. A 15% masking rate results in a task of unmasking too few tokens. Consequently, we need to mask a higher proportion of words. We follow the intuition that the number of words and the number of masked words should be related. While we want to focus on challenging cases, we must also create contexts containing nearly all words since such context will occur during inference. For a function name with n words, we define the number of masked words, from 0 to n, as a random variable M, which follows the discrete probability distribution $\mathit{Softmax}(C)$, where C is a vector of dimension $n+1$ with $C_i = 1 + i/n$. In our MLM task, M words are thus masked randomly among all possible subsets.

For instance, consider the four-word function name `convert_hex_to_int`. The number of masked words M follows the distribution $\mathit{Softmax}([1, 1.25, 1.5, 1.75, 2]) \approx [0.114, 0.147, 0.188, 0.241, 0.310]$. The formula ensures that masking all words is the most probable event but maintains a decent probability of masking a few words. If two words happen to be masked, then a subtask could be to recover `convert_hex_to_int` from `[M]_hex_[M]_int` given function tokens described in §4 as extra context.

5.2 Flexible Autoregression

LORD is not limited to left-to-right inference. It could decode the function name in one step [54, 62]. However, decoding the whole name in one step introduces difficulties in inferring relations between words, as the decoder better approximates the conditional distribution of a word given context [64]. As the example shows in Figure 4, the decoder might initially predict `int` for both positions 2 and 3. However, once `int` is placed at position 3, the likelihood of `int` also being at position 2 becomes very low.

Hence, we introduce a new autoregressive process called the flexible autoregression. It starts with an initial prediction containing only masked words. At each inference step, it runs our decoder on the current prediction.[1] The decoder estimates the possible words at each masked position. The autoregression picks the word w with the overall highest confidence score (probability) at any given position. It stops if that confidence score is smaller than the confidence threshold T or if all words are unmasked. Otherwise, it replaces the corresponding masked word with the word w, and it continues. The confidence threshold T is a parameter to flexible autoregression for addressing distribution shifts between the training and test sets, thereby improving the precision. We select the threshold that achieves the best F_1 score on the validation set.

We give a toy example of the process in Figure 4. It first infers the `[EOS]` word; as explained before this special word is masked from the decoder context. It continues by positioning the `int` word in the fourth position because the confidence score at the third position is lower (0.29 vs 0.35), leaving some positions for earlier words. Then, it notices an array that contains, in fact, a hexadecimal number, so it adds the `hex` word at the second position. It concludes correctly that the function is a conversion from a hexadecimal number to an integer by correctly placing the preposition `to` and the verb `convert` in two steps.

[1]Because `[EOS]` words are never part of the context during our MLM task, we have to mask them from the decoder.

6 Evaluation

In this section, we evaluate BLens comprehensively. In §6.1, we describe the evaluation methodology, including the dataset, metrics for function name prediction, and competitors. With this methodology, we evaluate BLens in the cross-binary setting (§6.2) and the cross-project setting (§6.3). In both settings, we investigate whether BLens outperforms competitors and analyze BLens predictions. Furthermore, we introduce a strict setting that removes shared components across projects to evaluate generalization capabilities (§6.4). We perform an ablation study on BLens to validate our contributions against other design choices (§6.5). The discussion in §7 indicates that, in the challenging cross-project setting, BLens's predicted function names can be meaningful even if different from the ground truth.

6.1 Methodology

Dataset. We use the Punstrip [56] dataset, as does XFL [58]. It contains 741,724 functions originating from 10,047 C binaries extracted from more than three thousand Debian packages[2], which have been pre-compiled with different compilers and compiler versions. Following previous work [56, 58], we exclude empty functions, overlapping functions, and locally bound symbols, and we use the ELF symbol table to obtain function names.

Tokenizer. To predict words in function names, we first have to transform function names into sequences of words. We use the tokenizer by Patrick-Evans et al. [58], which assembles a vocabulary of 1024 words, along with a set of words for each function name. We wrote a recursive version of the tokenizer algorithm to produce sequences instead of sets.

Settings. We adopt the two settings defined by Xiong et al. [83]: the cross-project setting and the cross-binary setting, to understand both practical performance and the model's generalization capability. In both settings, we split functions into training, validation, and test sets, which comprise approximately 80%, 10%, and 10% of the total dataset, respectively. In the **cross-project** setting, an entire project is allocated to one of the sets, ensuring that functions from the same project belong to the same set. In the **cross-binary** setting, we only ensure that functions from the same binary file belong to the same set. The cross-project setting is more realistic since,

[2]For simplicity, we refer to packages as projects in this paper.

in most use cases, a reverse engineer is working on an unseen project. However, even different projects may reuse code from certain APIs, design patterns, and templates, which is also expected in a real-world deployment of function name prediction. To specifically evaluate generalization abilities, we define the **strict setting** on top of the cross-project setting, which removes potential shared components across projects.

Pre-processing. As explained in §4, BLens employs two function embeddings, DEXTER [58] and CLAP [75], plus one basic block embedding, PALMTREE [44]. All embeddings employ their own pre-processing, tied to specific disassemblers. DEXTER uses the radare2 [71] (v2-5.5.4) disassembler. Because it relies heavily on vocabularies extracted from the training set (e.g., external calls, operations), we had to train distinct DEXTER models for the cross-binary and cross-project settings. Note that DEXTER reads function boundaries from the symbol table because it considers recovering function boundaries from stripped binary code an orthogonal task. To obtain CLAP and PALMTREE embeddings, we employ pre-trained models from their authors. We use IDA Pro (v7.6) as a disassembler to obtain CLAP function embeddings. Likewise, we compute PALMTREE instruction embeddings with angr [66] (v9.2.93) as a disassembler. We average the instruction embeddings of each basic block to obtain sequences of basic block embeddings for each function.

Metrics. Let us define the list of predicted function names $\hat{X}_1, \ldots, \hat{X}_B$ and the list of ground truth function names $X_1, \ldots, X_B$. Metrics commonly used for function name prediction consider $\hat{X}_i$ as a set of words $\{\hat{x}_1, \ldots, \hat{x}_j\}$, while the ground truth X_i is another set $\{x_1, \ldots, x_k\}$. True positives for the i th function (TP_i), false positives (FP_i), and false negatives (FN_i) are defined as:

$$TP_i = |\{\hat{x}_i\} \cap \{x_i\}|, FP_i = |\{\hat{x}_i\} \setminus \{x_i\}|, FN_i = |\{x_i\} \setminus \{\hat{x}_i\}|$$

Because it gives equal weight to each word, we adopt the micro-average definition of the precision P and recall R:

$$P = \frac{\sum_{i=1}^{N} TP_i}{\sum_{i=1}^{N}(TP_i + FP_i)}, \quad R = \frac{\sum_{i=1}^{N} TP_i}{\sum_{i=1}^{N}(TP_i + FN_i)}$$

Recall is fundamental to offer a decent number of predictions. Moreover, precision is equally crucial in our context, since false positives can only be discovered after complex human analysis of assembly code. Hence, a good balance between recall and precision is necessary for function name prediction. The F_1 score measures this balance by the harmonic mean of precision and recall.

Word order metrics. Nevertheless, F_1 does not consider the order of words or any repetitions of words in the ground truth. Therefore, we also consider two classical metrics for image captioning and translation that operate on n-grams. The first metric, ROUGE-L [49], referred to as RougeL in this paper, measures the longest common subsequences and thus places a greater emphasis on recall. The second metric, the smoothed version of BLEU-4 [50], referred to as Bleu in this paper, focuses on n-gram precision scores. Despite the application of a brevity penalty, in the end, it remains a precision metric. Therefore, although these metrics are crucial for capturing word order, F_1 still remains the primary metric of function name prediction.

Embedding metrics. As an alternative to classical metrics, embedding-based metrics, such as BERTSCORE [93] and VarCLR [12], promise to measure semantic similarity between predictions and ground truth with the distance between their embeddings. In our evaluation, we include measurements of VarCLR, as it is trained on software and designed for variable names. However, it comes with inherent limitations (at times, counter-intuitive judgments and issues with empty predictions; see §7), which is why we emphasize the need for F_1 as a robust performance metric.

Free functions. To accurately simulate a real-world scenario, twenty functions that can be inferred automatically by static analysis (e.g., `csu_init`) are treated as perfectly predicted in the cross-binary and cross-project settings. This function list is adopted from Patrick-Evans et al.'s evaluation of XFL [57] and reported in Appendix A.

Implementation. BLens is implemented in Pytorch and utilizes parts of CoCa and GIT implementations. All experiments were run in two weeks on a server with access to one terabyte of main memory and four NVIDIA H100 GPUs, each with 80GB of VRAM. The BLens transformer architecture features a token dimension d of 768 and a batch size of 512. The number of function patches k_1 and image tokens k_2 are 82 and 64, respectively. Function names consist of up to 20 words. BLens transformer blocks use multi-head attention layers with 32 heads of dimension 24. The large number of heads is effective for handling multiple embeddings, while the relatively low dimension helps avoid overfitting. The pre-trained model contains 153 million learnable parameters, decreasing to 138 million following fine-tuning. Further implementation details are documented in our artifact [7].

Training. Both the pre-training and fine-tuning phases last 200 epochs. Every 10 epochs during fine-tuning, confidence thresholds are evaluated to find the optimal threshold according to the F_1 score on the validation set. The model is then saved along with its threshold. In the ablation study, each phase has 80 epochs, and we obtain a confidence threshold as well as a model every 4 epochs. Eventually, the best model on the validation set is evaluated over the test set using the corresponding confidence threshold. The confidence threshold turned out to be 0.194 for the cross-project setting and 0.398 for the cross-binary setting.

Competitors. We consider three main competitors for the function name prediction task. AsmDepictor by Kim et al. [37] translates instruction sequences to function names with a transformer. SymLM by Jin et al. [35] is a transformer that considers both the target function and the function calling context. External calls are embedded, while internal functions

Model	F_1	RougeL	Bleu	VarCLR
BLens	**0.772**	**0.699**	**0.582**	**0.819**
XFL	0.625	0.527	0.325	0.720
BL-S	**0.789**	**0.689**	**0.652**	**0.817**
SymLM	0.704	0.472	0.107	0.736
BL-A	**0.763**	**0.693**	**0.596**	**0.819**
AsmDep.	0.407	0.432	0.279	0.649

Table 1: Results in the cross-binary setting. BL-A and BL-S are variants trained on functions obtained from SymLM and AsmDepictor pre-processing.

Word	Occurrences	Prec.	Recall	F_1
`ocaml`	1852	0.983	0.990	0.987
`get`	**1350**	**0.864**	**0.740**	**0.797**
`set`	**687**	**0.880**	**0.702**	**0.781**
`string`	**683**	**0.898**	**0.783**	**0.836**
`2`	550	0.815	0.867	0.840
`4`	418	0.995	0.965	0.980
`fun`	403	0.681	0.492	0.571
`parse`	378	0.902	0.595	0.717
`to`	365	0.868	0.660	0.750
`read`	**362**	**0.881**	**0.709**	**0.786**
`print`	**361**	**0.842**	**0.738**	**0.787**
`initialise`	360	0.806	0.613	0.696
`reiser`	351	0.997	0.989	0.993
`list`	**340**	**0.888**	**0.572**	**0.696**
`free`	**326**	**0.920**	**0.837**	**0.876**
`is`	317	0.840	0.698	0.762
`buffer`	**314**	**0.932**	**0.722**	**0.814**
`lwt`	273	0.996	0.424	0.595
`name`	252	0.896	0.677	0.771
`integer`	**250**	**0.887**	**0.779**	**0.830**

Table 2: Top 20 predicted words in the cross-binary setting. Functions given for free (described in §6.1) are discarded.

are represented by Trex embeddings [60]. XFL by Patrick-Evans et al. [58] predicts word sets with PFastreXML [32] and DEXTER before ordering them with an n-gram model.

Adaptation. Our competitors use different function name tokenizers; thus, we adapted each to work with our tokenizer and trained them in both the cross-binary and cross-project settings for a fair comparison. With their pre-processing phase, BLens and XFL gathered 436,941 functions, while SymLM and AsmDepictor gathered 232,393 and 318,372 functions, respectively. Due to these differences, we had to train specific versions of BLens on SymLM and AsmDepictor datasets for both the cross-binary and cross-project settings. We call the first model BL-S and the second BL-A.

HexT5. HexT5 [83] is also an interesting competitor; however, the source code for the learning phase and data processing is not available. Therefore, we could only reimplement the binary processing step based on the provided input examples and use the pre-trained model. This model uses a vastly different tokenizer. Consequently, we conduct a qualitative evaluation of HexT5, which is discussed further in §7.

6.2 Cross-Binary Setting

Results in the cross-binary setting are summarized in Table 1. BLens outperforms XFL by 23.5%, 32.6%, 79.1%, and 13.7% in terms of F_1, RougeL, Bleu, and VarCLR scores, respectively. BL-S outperforms SymLM by 12%, 45.9%, 507%, and 11.1% in terms of F_1, RougeL, Bleu, and VarCLR scores, respectively. BL-A outperforms AsmDepictor by 87.7%, 60.5%, 113%, and 26.1% in terms of F_1, RougeL, Bleu, and VarCLR scores, respectively. BLens obtains significantly better metrics, especially RougeL and Bleu scores. Moreover, BLens achieves an F_1 score of 0.772 due to a high precision of 0.917 and a good recall of 0.666. BLens RougeL, Bleu, and VarCLR scores are 0.699, 0.582, and 0.819, which suggest a good capacity to recover subtle grammar details and produce meaningful names.

In Table 2, we report the most predicted words of BLens in the cross-binary setting, along with their precisions, recalls, and F_1 scores. Note that functions given for free (described in §6.1) are discarded from scores. Note further that BLens is nearly perfect at predicting words specific to `ocaml` and certain libraries, such as `reiser`. These words occur as prefixes in many functions and are straightforward to recover: OCaml code has distinct patterns, and ReiserFS-specific functions are prefixed by `reiser4`, which is part of the reason we also achieve a very high score on the word 4. Moreover, some standard OCaml functions are present in multiple binaries, and ReiserFS functions are shared across three binaries (prefixes and code sharing are specifically addressed in §6.4).

BLens also achieves good results on other words. For instance, on `get` and `set`, which are common general-purpose terms, BLens attains F_1 scores of 0.797 and 0.781, respectively. On words related to strings and input/output operations, such as `string`, `parse`, `read`, and `print`, BLens achieves F_1 scores of 0.836, 0.717, 0.786, and 0.787, respectively. Finally, on words related to low-level operations such as `initialise`, `free`, `list`, and `buffer`, BLens achieves F_1 scores of 0.696, 0.876, 0.696, and 0.814, respectively.

Conclusion. In the cross-binary setting, BLens achieves a notable F_1 score of 0.771, with a critical precision of 0.917. Thanks to our contributions, BLens captures word order, domain-specific words, and general-purpose words, improving the state of the art by 12% on F_1, 32.6% on RougeL, 79.1% on Bleu, and 11.1% on VarCLR scores.

Model	F_1	RougeL	Bleu	VarCLR
BLens	**0.460**	**0.393**	**0.242**	**0.648**
XFL	0.295	0.222	0.025	0.560
BL-S	**0.394**	**0.294**	**0.221**	**0.599**
SymLM	0.277	0.171	0.023	0.534
BL-A	**0.455**	**0.399**	**0.258**	**0.652**
AsmDep.	0.200	0.220	0.089	0.520

Table 3: Results in the cross-project setting.

6.3 Cross-Project Setting

Results in the cross-project setting are summarized in Table 3. BLens outperforms XFL by 55.7%, 76.7%, 887%, and 15.9% in terms of F_1, RougeL, Bleu, and VarCLR scores, respectively. BL-S outperforms SymLM by 42.2%, 71.7%, 855%, and 12.1% in terms of F_1, RougeL, Bleu, and VarCLR scores, respectively. BL-A outperforms AsmDepictor by 128%, 81.1%, 188%, and 25.3% in terms of F_1, RougeL, Bleu, and VarCLR scores, respectively.

Again, BLens achieves significantly better metrics, especially F_1, RougeL, and Bleu scores. Moreover, BLens achieves an F_1 of 0.460 due to a good precision of 0.655 and a decent recall of 0.354. BLens RougeL, Bleu, and VarCLR scores are 0.393, 0.242, and 0.648 which suggest a decent capacity to recover word order and produce meaningful function names.

In Table 4, we report the most predicted words of BLens in the cross-project setting, along with their F_1 scores. Again, functions given for free (described in §6.1) are discarded. Even in the cross-project setting, the extremely high scores on some words (e.g., an F_1 of 0.960 for the word `visit`) can be attributed to the training set being highly relevant to a new target program. BLens obtains nearly perfect scores for the word `ocaml` because our dataset includes approximately 80 OCaml libraries as projects. Additionally, the source code of the OCaml standard library is shared across three projects, while individual OCaml libraries employ this code base, highlighting interconnections between projects and shared code components. Likewise, the word `curry` is frequently used by the OCaml compiler. Similarly, BLens achieves impressive metrics for the word `visit`. This word appears in both the test and training sets due to the common reliance on the QAPI interface. The QAPI (QEMU API) schema defines a set of data types explored with the visitor pattern. For the word `soap`, the project gridsite-clients in the test set uses the SOAP (Simple Object Access Protocol) for web service communications, a pattern similarly observed in the training data from the parent project gridsite and the related project lfc. Lastly, BLens can accurately predict the word `usal` thanks to functions from the libusal library inside projects like genisoimage and wodim. The libusal library supports creating CD and DVD images.

	Cross-Project		Strict	
Word	Occ.	F_1	Occ.	F_1
`ocaml`	2162	0.975	0	-
`get`	1132	0.365	813	0.313
`string`	**1050**	**0.498**	**817**	**0.465**
`free`	**519**	**0.750**	**379**	**0.688**
`type`	**465**	**0.780**	**150**	**0.599**
`initialise`	435	0.352	372	0.305
`fun`	**401**	**0.631**	**365**	**0.640**
`set`	392	0.336	307	0.295
`visit`	389	0.960	0	-
`soap`	337	0.998	0	-
`print`	**332**	**0.486**	**290**	**0.434**
`read`	326	0.418	311	0.441
`2`	305	0.212	207	0.110
`curry`	304	1.000	0	-
`path`	288	0.368	279	0.378
`name`	285	0.417	255	0.388
`usal`	266	0.981	0	-
`error`	**263**	**0.675**	**208**	**0.601**
`open`	**254**	**0.556**	**238**	**0.584**
`information`	244	0.650	58	0.205

Table 4: Top 20 words predicted in the cross-project setting with their occurrences and F_1 scores in the cross-project and strict settings.

We can categorize the rest of the words into different operation groups. Firstly, words related to data manipulation include `get`, `set`, `read`, `name`, and `path`. BLens achieves F_1 scores of 0.365, 0.336, 0.418, 0.417, and 0.368, respectively. Secondly, for memory management, words include `free`, `initialise`, and `open`. BLens exhibits good performance with F_1 scores of 0.750, 0.352, and 0.556, respectively. Notably, `free` shows a high F_1 score, demonstrating BLens's ability to identify memory liberation. Thirdly, on words related to output and notification, such as `print` and `error`, BLens obtains F_1 scores of 0.486 and 0.675. The higher F_1 score for `error` indicates good performance in identifying error handling code. Lastly, on miscellaneous words such as `string`, `type`, `fun`, and `information`, BLens obtains F_1 scores of 0.498, 0.780, 0.631, and 0.650, respectively.

Conclusion. In the cross-project setting, BLens significantly outperforms SotA methods, particularly on metrics capturing word order, with 42.2% F_1, 71.7% RougeL, 188% Bleu, and 12.1% VarCLR score improvement. BLens achieves notable F_1 scores on some general-purpose words, for instance, 0.750 for `free` and 0.675 for `error`. While 0.460 F_1, 0.393 RougeL, and 0.24 Bleu scores seem low in absolute terms, predictions by BLens may be meaningful even when different from the ground truth, as we discuss in §7.

Model	F_1	RougeL	Bleu	VarCLR
BLens	**0.294**	**0.243**	**0.094**	**0.568**
XFL	0.085	0.055	0.001	0.474
BL-S	**0.299**	**0.240**	**0.110**	**0.571**
SymLM	0.195	0.134	0.011	0.518
BL-A	**0.321**	**0.272**	**0.098**	**0.585**
AsmDep.	0.090	0.085	0.034	0.445

Table 5: Results in the strict setting.

6.4 Strict Setting

This experimental setting extends the cross-project setting to rigorously evaluate generalization capabilities by minimizing potential shared code components across projects. Ideally, this would involve deduplication at the source code level; however, as the dataset consists of pre-compiled binaries, we define a set of heuristics derived from manual analysis to identify and remove as many similar components from binaries as possible. Initially, we remove hash duplicates and, for completeness, also freely given functions (see §6.1). We observe a relatively consistent drop of about 0.059 in the F_1 score across most methods. However, the impact is minimal on SymLM and BL-S, as SymLM's pre-processing has already removed part of these functions.

To identify shared code where the hash differs after compilation, we analyze the words in function names. Note that we cannot simply remove all functions from testing whose names occur in the training set, as this would exclude many benign cases such as `main` or `print_help`. Shared code typically arises from statically-linked runtime functions and library calls, which often share name prefixes (e.g., `ocaml`, `soap`, `usal`), or non-prefix words that frequently co-occur with prefixes (e.g., `curry` co-occurs with `ocaml`, as it denotes curried functions). We generate an exclusion list of prefixes and co-occurring words found from a semi-automatic analysis of the top 200 performing words and binaries. Any function whose name appears in the training set, contains an excluded word, and has an $F_1 > 0$, is removed from the test set. Allowing functions with an F_1 of 0 avoids removing some functions containing excluded words by chance. From the remaining function names in the test set, we remove all excluded words.

Ultimately, 40 words and 6491 functions (2644 freely given functions, 2526 hash duplicates, and 1321 other duplicates) are removed from 1024 words and 23,875 initial functions in the test set. In Table 4, we report the new occurrences and F_1 scores of the previously most predicted words in the strict setting. We observe that only general purpose words such as `string`, `free`, and `type` remain, with F_1 scores of 0.465, 0.688, and 0.599, respectively. Results are summarized in Table 5. While BLens's F_1 score decreases from 0.460 to 0.294, it still outperforms all competitors.

Model	F_1	RougeL	Bleu	VarCLR
BLens	**0.445**	**0.376**	**0.207**	**0.639**
BL-NP	0.287	0.257	0.057	0.554

Table 6: COMBO ablation results. BL-NP is a variant trained without a pre-training phase. Ablation models are trained for only 80 epochs.

Ensemble	F_1	RougeL	Bleu	VarCLR
C+P+D	**0.445**	0.376	**0.207**	**0.639**
C+D	0.438	**0.378**	0.196	0.638
C+P	0.425	0.370	0.182	0.629
P+D	0.364	0.293	0.096	0.595
C	0.425	0.366	0.184	0.633
P	0.352	0.293	0.109	0.593
D	0.310	0.254	0.060	0.574

Table 7: Input embeddings ablation results. C: CLAP, P: PALMTREE, D: DEXTER, +: Combination of embeddings.

Conclusion. In the strict setting, BLens continues to significantly outperform the state of the art, with 53.3% F_1, 79.4% RougeL, 188% Bleu, and 10.3% VarCLR improvement. The widening of the gap between BLens and other methods in terms of F_1 highlights BLens's generalization.

6.5 Ablation Study

We now evaluate the impact of each component of BLens to show that they are all beneficial. We start by evaluating the impact of COMBO, which aligns multiple embeddings together in a joint space with function names during pre-training. Then, we assess the contribution of each input embedding (CLAP, PALMTREE, and DEXTER) by evaluating each subset of these embeddings. Finally, we investigate the advantages of LORD over the traditional decoder architecture and the direct use of COMBO as a decoder. We conduct the ablation in the cross-project setting with training phases of 80 epochs, optimizing the confidence threshold every four epochs.

6.5.1 COMBO

We define a new model, BL-NP, which is trained without the COMBO pre-training phase. This model starts with the ensemble encoder and turns embeddings into patches. Then, instead of using the function encoder pre-trained on two tasks, this model simply passes these patches to LORD function name decoder for fine-tuning. In Table 6, we report metrics of BL-NP and the base model BLens. BLens significantly outperforms BL-NP across all metrics, with F_1 and VarCLR scores of 0.445 and 0.639, compared to BL-NP 0.287 and 0.554,

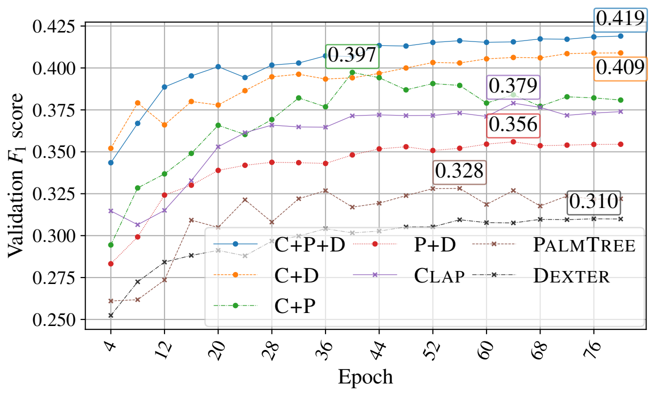

Figure 5: Curve of validation F_1 scores over the fine-tuning for the input embeddings ablation models.

respectively. This higher performance is further reflected in the RougeL and Bleu scores, which are 0.376 and 0.207 for BLens, compared to 0.257 and 0.057 for BL-NP, respectively.

Conclusion. The F_1 score gain of 0.158 clearly demonstrates the relevance of the contrastive captioning task.

6.5.2 Input Embeddings

Thanks to COMBO, BLens incorporates multiple embeddings. The original model used CLAP, PALMTREE, and DEXTER. Now, we train models with all possible combinations of these embeddings. We report the metrics of each model in Table 7. We remark that CLAP is the most effective embedding, evidenced by its F_1 score of 0.425, compared to PALMTREE and DEXTER, which have F_1 scores of 0.352 and 0.310, respectively. Combining all three embeddings achieves the best F_1 score of 0.445. Although the improvement over the combination of CLAP and DEXTER amounts to merely 0.007 F_1 score, an analysis in Figure 5, reveals a consistent gain of around 0.01 on the validation set during fine-tuning.

Conclusion. While CLAP embeddings are powerful, the synergy of CLAP, PALMTREE, and DEXTER embeddings through COMBO is the most effective design.

6.5.3 LORD

LORD introduces a novel MLM task during fine-tuning and a confidence threshold during inference to enhance precision. Thus, we devise a version of BLens named SIMPLE, employing teacher forcing during fine-tuning with traditional left-to-right autoregression, but still with a confidence threshold. Additionally, we assess LORD-T0, a version of LORD where the confidence threshold is fixed at 0. SIMPLE-T0 combines these two modifications. Lastly, as the COMBO pre-training incorporates a captioning task, we also evaluate COMBO's direct predictions. We report summary results in Table 8. COMBO as a decoder performs the worst across

Decoder	Prec.	Recall	F_1	RougeL	Bleu	VarCLR
LORD	**0.667**	0.334	**0.445**	0.376	0.207	0.639
SIMPLE	0.471	0.379	0.420	0.400	0.235	0.651
LORD-T0	0.436	0.377	0.404	0.397	**0.261**	0.642
SIMPLE-T0	0.427	**0.392**	0.409	**0.411**	0.242	**0.653**
COMBO	0.496	0.278	0.357	0.308	0.154	0.602

Table 8: LORD ablation results. LORD: BLens decoder, SIMPLE: Standard left-to-right decoder, -T0: Confidence threshold is zero, COMBO: COMBO as a decoder.

all metrics. This is expected; while COMBO pre-training benefits from multiple objectives [8], conflicts and imbalances during optimization can hinder the performance of individual objectives [65]. Fine-tuning is therefore essential for downstream tasks.

LORD achieves the highest precision at 0.667 and F_1 score at 0.445, demonstrating the effectiveness of both the novel MLM task and the confidence threshold despite a trade-off with the recall. In contrast, SIMPLE, LORD-T0, and SIMPLE-T0 achieve better recall and RougeL scores, as RougeL is recall-related. Moreover, SIMPLE-T0 achieves the highest VarCLR score at 0.653 and LORD-T0 the highest Bleu at 0.261. Nevertheless, the contributions of LORD are positive as the F_1 score is the primary metric (see §6.1).

Conclusion. LORD allows an F_1 score gain of 0.036 compared to the usual decoder architecture, thanks to a significant precision gain of 0.24, which is essential in real-life scenarios. Moreover, COMBO pre-training alone leads to weak results.

7 Discussion

The evaluation of BLens shows significant improvement over state-of-the-art methods in both cross-binary and cross-project settings. Moreover, an ablation study confirms the relevance of our contributions to BLens's success.

Despite these successes, the F_1 score of 0.460 in the cross-project setting is relatively low in absolute terms, warranting further investigation. To investigate this issue further, we perform a qualitative case study on six instances where BLens predictions diverged from the ground truth. This analysis aims to determine whether such predictions can be meaningful and contextually accurate despite not matching the ground truth. For completeness, we also include predictions from XFL, SymLM, AsmDepictor, and HexT5 in the same cases. Predictions, ground truth, and function-level scores are shown in Table 9. We analyze the corresponding source code in each case to evaluate if the predictions are appropriate. For reference, we include source code for the functions in Appendix B.

We first discuss the predictions of BLens before briefly assessing those of other methods:

Case	Name	Ground truth	Model	Prediction	F_1	RougeL	Bleu	VarCLR
❶	`execute`	`execute`	BLens	`shell`	0	0	0.841	0.521
			XFL	`___`	0	0	0	0.564
			SymLM.	`file_main`	0	0	0.639	0.413
			AsmDep.	`string_at`	0	0	0.639	0.414
			HexT5	`print_sme_za_list`	0	0	0.302	0.334
❷	`evalbackcmd`	`evaluate_back_cmd`	BLens	`pipe`	0	0	0.114	0.336
			XFL	`___`	0	0	0	0.485
			SymLM	`read`	0	0	0.114	0.377
			AsmDep.	`next_window`	0	0	0.388	0.408
			HexT5	`new_logical_line`	0	0	0.452	0.389
❸	`fsa_get_config`	`get_config`	BLens	`key_file_dump`	0	0	0.452	0.427
			XFL	`___`	0	0	0	0.417
			SymLM	`___`	0	0	0	0.417
			AsmDep.	`search_done`	0	0	0.639	0.381
			HexT5	`madacc_size`	0	0	0.639	0.264
❹	`dtwstime`	`time`	BLens	`print_date`	0	0	0.639	0.594
			XFL	`___`	0	0	0	0.536
			SymLM	`time`	1	1	1	1
			AsmDep.	`cli_color`	0	0	0.639	0.321
			HexT5	`reset_items`	0	0	0.639	0.361
❺	`removeFromEdited`	`remove_from`	BLens	`text_delete`	0	0	0.639	0.696
			XFL	`___`	0	0	0	0.548
			SymLM	function absent from the dataset	-	-	-	-
			AsmDep.	`directory`	0	0	0.309	0.437
			HexT5	`_collector_env_save_preloads`	0	0	0.302	0.361
❻	`task_panel_cancel_clicked_cbk`	`task_panel_cancel_clicked_callback`	BLens	`on_button_cancel_clicked`	0.444	0.444	0.368	0.756
			XFL	`___`	0	0	0	0.276
			SymLM	`___`	0	0	0	0.276
			AsmDep.	`___`	0	0	0	0.276
			HexT5	`gsl_sf_lnfact_e`	0	0	0.235	0.153

Table 9: Case study of predictions in the cross-project setting. '`___`': Empty prediction. We report function-level F_1, RougeL, Bleu, and VarCLR scores. Bleu scores are surprisingly high for predictions that differ from the ground truth, because the smoothed version of BLEU-4 [50] inflates n-gram precisions, which renders the score meaningless for very short sentences. In the same way, all of F_1, RougeL, and Bleu scores for HexT5 are affected by HexT5 employing a different vocabulary.

❶ BLens predicts `shell` instead of `execute`. The source code executes a shell script given in argument, therefore the prediction is close to the actual behavior of the function.
❷ BLens predicts `pipe` instead of `evaluate_back_command`. Since the source code executes a command inside back quotes with a pipe, BLens has captured a key functionality.
❸ BLens predicts `key_file_dump` instead of `get_config`. In the source code, the configuration returned is an instance of the GKeyFile structure. Hence, this prediction is more precise than the ground truth.
❹ BLens predicts `print_date` instead of `time`. Although the function does not print, it returns a structure akin to a datetime, making the prediction useful.
❺ BLens predicts `text_delete` instead of `remove_from`. The original name of the function was `removeFromEdited`, but `edited` is not part of the vocabulary. This function comes from the hexedit project, a hexadecimal and ASCII text editor. The source code reveals that this function deletes text from an existing edition; therefore, the prediction is appropriate. Despite an F_1 score of 0, VarCLR validates the prediction with a score of 0.696.
❻ BLens predicts `on_button_cancel_clicked` instead of `task_panel_cancel_clicked_callback`. Although different from the ground truth, this prediction is semantically close, as reflected by a VarCLR score of 0.756. The name pattern `on...clicked` is common for callbacks on click events in graphical user interfaces, and the choice of `button` instead of `task_panel` can be explained by the function's first argument being a `GtkButton` that has been clicked on.

In case six, BLens generates a clever rephrasing of the ground truth. The ablation study data allows us to assess each component of BLens qualitatively. Without COMBO pre-training or with a single embedding, BLens predicts nothing. Combining CLAP with PALMTREE results in `ui`, and with CLAP and DEXTER in `on_delete_clicked`. A simpler decoder yields `on_entry_activate`. This highlights the effectiveness of BLens's design.

Now, we briefly assess the other methods. Firstly, XFL predictions are entirely empty, indicating that the model is conservative and avoids making incorrect predictions in challenging examples, resulting in more false negatives but fewer false positives. Secondly, among SymLM's five predictions, two are inappropriate based on the function names and source code (e.g., ❷: `read` instead of `evaluate_back_command`), two are empty and one is correct. Note that in case five, the function is absent from SymLM's sub-dataset. Thirdly, AsmDepictor provides six predictions, of which five are inappropriate (e.g., ❹: `string_at` instead of `execute`), and one is empty. Lastly, all six predictions from HexT5 are inappropriate and meaningless (e.g., ❺: `_collector_env_save_preloads` instead of `remove_from`). A quick review of more predictions reveals that the pre-trained HexT5 model, which we used due to the unavailability of HexT5 training implementation, performs very poorly. This could also be due to large distribution shifts between datasets and low generalization from the model.

Overall, this case study suggests that, contrary to other methods, BLens can predict meaningful names in the cross-project setting even if the F_1 score is low. Nevertheless, there are some further points to discuss.

Dataset overlap. Concerns about dataset overlap arise because we reused pre-trained CLAP and PALMTREE embeddings. However, these pre-trained models are based on broad datasets and trained with other learning objectives. Even if they have encountered some projects from our dataset, they are designed to generalize across different projects and compiler settings, rather than memorizing specific examples. Using pre-trained embeddings is common in machine learning as it leverages general-purpose models for new tasks, saving computational resources and enhancing model robustness for specific fine-tuning tasks. Nevertheless, BLens already surpasses the state of the art with only DEXTER embeddings and merely 80 epochs (Table 7).

Dataset diversity. Our dataset is taken from related research [56, 58]. This choice enables straightforward comparisons and grants a tokenizer specifically designed for this dataset. Since this dataset consists mostly of binaries from Debian packages written in C, the three thousand projects lack diversity. Nevertheless, BLens could be applied to other datasets and programming languages.

Evaluation metrics. Evaluating function names remains challenging due to the complexity of natural language. VarCLR measures semantic similarity by embedding names to a high-dimensional space. This captures a degree of semantic similarity, as illustrated by cases five and six. Yet, VarCLR assigns high scores to empty predictions because the empty string lies at the center of the space. As seen in Table 9, VarCLR favors empty predictions over meaningful alternatives in the first two cases. Additionally, incorrect predictions appear relevant (e.g., `directory` scores 0.437 for `remove_from`), and VarCLR can be misaligned with human judgment, preferring `next_window` over `pipe` for `eval_back_cmd`. In contrast, the F_1 score relies strictly on the set of words in the ground truth. Pre-processing of function names partially normalizes the ground truth, successfully addressing most abbreviations and smaller variations.

CodeWordNet [35, 83] proposes considering word equivalences (e.g., `initialize` and `create`) during the F_1 computation. However, this requires manual validation and oversimplifies context-specific synonyms. An alternative method is to use source code to better normalize function names, as proposed by Carvalho et al. [9]. For example, in the 3dchess project, `dir` would expand to `direction` rather than the more common `directory`. However, this requires access to the training data source code, which was not feasible in this study.

8 Related Work

Function name prediction. Pioneering work on function name prediction relies on traditional machine learning techniques. For instance, Debin [28] and Punstrip [56] employ probabilistic graphical models, while NRFE [24] is a lightweight framework over various features. These approaches have low granularity and de facto identify functions frequent in the training data. On the other hand, XFL [58] predicts sets of words with multi-label classification. However, XFL's word ordering is agnostic to the binary code. Recent works use the transformer [72] encoder-decoder architecture to translate binary code into function names, treating binary code and human-readable languages both as natural languages. For instance, AsmDepictor [37] focuses on instructions, NERO [15] uses augmented control flow graphs to represent the calling context, SymLM [35] employs pre-trained function embeddings to capture execution behavior, and HexT5 [83] works with low-level pseudocode, though it relies on a challenging decompilation step to convert from binary code.

Binary code information recovery. Variable name recovery and type inference require a more concrete representation of function behavior. The pioneering work Debin [28] relied on memory cells and instructions to recover variable names. Some recent approaches rely on decompiled pseudocode. DIRE [41], DIRECT [55], and DIRTY [11] use encoder-decoder architectures starting from pseudocode to recover variable names. The trend is toward designing multiple pre-training tasks [11, 83]. We already mentioned HexT5 [83], which unifies multiple tasks, including summarization and function name prediction. These approaches rely on pseudocode, thus departing from binary code. CP-BCS [86] incorporates control flow graphs and assembly code but still primarily uses pseudocode.

Source code information recovery. Efforts have also been made to recover source code information. Code2vec [1] learns function embeddings from abstract syntax trees and can be fine-tuned for various tasks such as code retrieval and classification. Interestingly, it achieves a good F_1 score on function name prediction across projects. CodeT5 [79] employs the transformer architecture to translate between function summaries and source code through various pre-training tasks. Source code presents different challenges than binary code because compiler optimizations introduce complex transformations and remove explicit types (e.g., unsigned int or array), making alignment harder.

Embeddings for binary functions. More approaches exist to represent binary functions. Xu et al. [85] use a Siamese architecture to train a graph neural network to represent control flow graphs, while Li et al. [47] use a deep graph network and learn graph editing distances. They learn to discriminate pairs of binary functions compiled from different sources. With the rise of transformer architectures, self-supervised tasks such as Masked Language Modeling have been adopted to learn deep semantics of assembly code [17, 76, 97].

Image captioning. We have discussed several pre-training tasks, including Contrastive Image-Language Pre-training (CLIP) [63] and contrastive captioning (CoCa) [88], which are essential to multimodal learning. Additional tasks have been developed, spanning unimodal tasks like image reconstruction [13] to multimodal tasks like object detection [46, 91], word-region alignment [13], and question answering [78]. Overall, the trend is toward combining multiple tasks in pre-training, a domain where OFA [78] excels. However, defining equivalent tasks for binary code is difficult. ViT [19], the original vision transformer, pioneered image patches and surpassed previous convolutional approaches [39, 82]. Since then, other transformers have scaled to hundreds of millions of parameters, avoiding overfitting due to the generality of images and massive datasets [30, 77].

Contrastive learning. Contrastive learning [27] minimizes the distance to anchors and maximizes it to others. We have already described techniques that employ text embedding as anchors, such as CLAP [75] and CONTRABIN [95]. BLens employs a learnable function name representation. Another line of work creates anchors with semantics-preserving transformations, but numerous transformations are required to obtain semantics-aware embeddings. Various transformations can be applied to source code [18, 33, 34]. Moreover, compiling the same source with different compilers, options, or for different architectures can produce anchors [34, 51, 70, 83].

Embeddings fusion. We use multimodal data fusion [96] to incorporate different views of functions. While fusing raw features [20, 73] might appear beneficial at first glance, fusing pre-trained embeddings takes advantage of the training and engineering efforts behind SotA embeddings. To fuse heterogeneous data, cross-modal and intra-modal attention mechanisms [20, 23, 31, 92], as well as a transformer [10, 42] can be applied. An efficient attention process requires a sufficient number of tokens. Our innovation involves cutting function-level embeddings into multiple smaller patches, enabling the model to capture finer-grained relationships within the data. By incorporating positional encoding, we maintain the sequential integrity of the data.

Transformer decoding. There are several other decoding schemes apart from the classic left-to-right paradigm [25, 45, 48, 62, 68, 69]. Our flexible autoregression scheme relies on Masked Language Modeling (MLM) to estimate the probability distribution of all unknown words given known words. Several parallel developments also utilize MLM to enhance speed [25] or to enable sentence generation in any order [48, 68].

9 Conclusion

We tackle the function name prediction problem by introducing BLens, a novel approach inspired by multimodal learning. It leverages two unique components: COMBO and LORD. COMBO first ensembles state-of-the-art binary function embeddings into function patches. Then, patches are projected into tokens aligned with the word latent space via contrastive captioning loss, thereby capturing spatial relationships within the binary code. LORD decodes the function tokens into the function name through a new Masked Language Modeling task tailored specifically to function names.

BLens sets a new state of the art in function name prediction, improving the main metric, F_1 score, by 12%, with a precision increase to 0.917, which is critical for real-world applications. Additionally, BLens achieves significant improvement in grammatical accuracy, with a 32.6% boost in RougeL and a 79.1% boost in Bleu scores. Against binaries coming from new projects, BLens demonstrates remarkable robustness, increasing F_1 by 42.2%, RougeL by 71.7%, and Bleu by an impressive 188%. Lastly, in a strict setting that reduces shared components BLens provides even more gains, with an improvement of 53.3% in terms of F_1 score.

Ethics Considerations

The dataset used in this study comes from open-source Debian packages, and we did not perform any unauthorized gathering of binary code. On the one hand, we acknowledge that our work in reverse engineering could have negative implications for some stakeholders, including its potential misuse for copyright infringement. On the other hand, advancing reverse engineering can significantly aid in identifying and mitigating malicious code early in an attack. Overall, we believe that our work provides a net positive impact on software security.

Open Science

We share the following artifacts on Zenodo [7]: BLens's implementation, pre-processed CLAP embeddings, PALMTREE embeddings, and DEXTER embeddings for our dataset.

Due to GPL version 3 [21] licensing constraints, we cannot share the binary Debian packages dataset, as it requires conjointly distributing the source code. However, the Debian binaries are freely available separately. We believe that the shared artifacts will support the reproducibility and validation of our research findings.

Acknowledgments

This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG), reference 378803395 (ConVeY) and France Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PETCC-0001.

References

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019.

[2] I. Arce. Bug hunting: The seven ways of the security samurai (supplement to comput. magazine). *IEEE Computer*, 35(04), 2002.

[3] F. Artuso, M. Mormando, G. A. Di Luna, and L. Querzoni. BinBert: Binary code understanding with a fine-tunable and execution-aware transformer. *IEEE Trans. Dependable Sec. Comput.*, 2024.

[4] L. J. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[5] G. Bachmann and V. Nagarajan. The pitfalls of next-token prediction. *arXiv preprint arXiv:2403.06963*, 2024.

[6] T. Baltrušaitis, C. Ahuja, and L.-P. Morency. Multimodal machine learning: A survey and taxonomy. *IEEE Trans. Pattern Anal. Mach. Intell.*, 41(2), 2018.

[7] T. Benoit, Y. Wang, M. Dannehl, and J. Kinder. BLens artifact. https://doi.org/10.5281/zenodo.14732394, 2025.

[8] R. Caruana. Multitask learning. *Mach. Learn.*, 28, 1997.

[9] N. R. Carvalho, J. J. Almeida, P. R. Henriques, and M. J. Varanda. From source code identifiers to natural language terms. *Journal of Systems and Software*, 2015.

[10] F. Chen, M. Han, H. Zhao, Q. Zhang, J. Shi, S. Xu, and B. Xu. X-llm: Bootstrapping advanced large language models by treating multi-modalities as foreign languages. *arXiv preprint arXiv:2305.04160*, 2023.

[11] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[12] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. Le Goues. VarCLR: Variable semantic representation pre-training via contrastive learning. In *IEEE/ACM Int. Conf. Software Engineering (ICSE)*. ACM, 2022.

[13] Y. Chen, L. Li, L. Yu, A. E. Kholy, F. Ahmed, Z. Gan, Y. Cheng, and J. Liu. UNITER: universal image-text representation learning. In *Proc. European Conf. Comput. Vision (ECCV)*. Springer, 2020.

[14] C. Cifuentes. The impact of copyright on the development of cutting edge binary reverse engineering technology. In *Working Conf. Reverse Engineering (WCRE)*. IEEE Comput. Society, 1999.

[15] Y. David, U. Alon, and E. Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

[16] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proc. Conf. North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. ACL, 2019.

[17] S. H. H. Ding, B. C. M. Fung, and P. Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE Symp. Security and Privacy (S&P)*. IEEE, 2019.

[18] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty. Towards learning (dis)-similarity of source code from program contrasts. In *Proc. Annu. Meeting Assoc. Computational Linguistics (ACL)*, 2022.

[19] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Int. Conf. Learning Representations (ICLR)*, 2021.

[20] M. Fang, S. Peng, Y. Liang, C.-C. Hung, and S. Liu. A multimodal fusion model with multi-level attention mechanism for depression detection. *Biomedical Signal Processing and Control*, 82, 2023.

[21] F. S. Foundation. GNU general public license version 3. https://www.gnu.org/licenses/gpl-3.0.html.

[22] D. Francis, P. A. Nguyen, B. Huet, and C. Ngo. Fusion of multimodal embeddings for ad-hoc video search. In

IEEE/CVF Int. Conf. Comput. Vision Workshops (ICCV Workshops). IEEE, 2019.

[23] D. Gao, K. Li, R. Wang, S. Shan, and X. Chen. Multi-modal graph neural network for joint reasoning on vision and scene text. In *Proc. IEEE/CVF Conf. Comput. Vision Pattern Recognition (CVPR)*. Comput. Vision Foundation / IEEE, 2020.

[24] H. Gao, S. Cheng, Y. Xue, and W. Zhang. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, 2021.

[25] M. Ghazvininejad, O. Levy, Y. Liu, and L. Zettlemoyer. Mask-predict: Parallel decoding of conditional masked language models. *arXiv preprint arXiv:1904.09324*, 2019.

[26] O. Goldman, A. Jacovi, A. Slobodkin, A. Maimon, I. Dagan, and R. Tsarfaty. Is it really long context if all you need is retrieval? towards genuinely difficult long context NLP. In *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 2024.

[27] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *IEEE Comput. Soc. Conf. Comput. Vision Pattern Recognition (CVPR)*. IEEE Comput. Society, 2006.

[28] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. T. Vechev. Debin: Predicting debug information in stripped binaries. In *Proc. ACM SIGSAC Conf. Comput. and Commun. Security (CCS)*. ACM, 2018.

[29] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conf. on Comput. Vision Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016.

[30] X. Hu, Z. Gan, J. Wang, Z. Yang, Z. Liu, Y. Lu, and L. Wang. Scaling up vision-language pre-training for image captioning. In *Proc. IEEE/CVF Conf. Comput. Vision Pattern Recognition (CVPR)*, 2022.

[31] Z. Hu, G. Feng, J. Sun, L. Zhang, and H. Lu. Bi-directional relationship inferring network for referring image segmentation. In *Proc. IEEE/CVF Conf. Comput. Vision Pattern Recognition (CVPR)*, 2020.

[32] H. Jain, Y. Prabhu, and M. Varma. Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In *ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 2016.

[33] P. Jain and A. Jain. Contrastive code representation learning. In *Proceedings of the 2021 Conf. on Empirical Methods in Natural Language Processing*, 2021.

[34] S. Jiang, C. Fu, S. He, J. Lv, L. Han, and H. Hu. Bincola: Diversity-sensitive contrastive learning for binary code similarity detection. *IEEE Trans. Softw. Eng.*, 2024.

[35] 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 *Proc. ACM SIGSAC Conf. Comput. and Commun. Security (CCS)*, 2022.

[36] L. Ke, W. Pei, R. Li, X. Shen, and Y.-W. Tai. Reflective decoding network for image captioning. In *Proc. IEEE/CVF Int. Conf. Comput. Vision (ICCV)*, 2019.

[37] H. Kim, J. Bak, K. Cho, and H. Koo. A transformer-based function symbol name inference model from an assembly language for binary reversing. In *Proc. ACM Asia Conf. Comput. and Commun. Security (ASIA CCS)*. ACM, 2023.

[38] J. F. Kolen and S. C. Kremer. *Dynamical Recurrent Networks*, pages 3–11. Wiley-IEEE Press, 2001.

[39] A. Kolesnikov, L. Beyer, X. Zhai, J. Puigcerver, J. Yung, S. Gelly, and N. Houlsby. Big transfer (bit): General visual representation learning. In *16th Europ. Conf. Comput. Vision (ECCV)*, volume 12350 of *LNCS*. Springer, 2020.

[40] A. Kristiadi, M. Hein, and P. Hennig. Being bayesian, even just a bit, fixes overconfidence in relu networks. In *Proc. Int. Conf. Machine Learning (ICML)*, volume 119 of *PMLR*, 2020.

[41] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig, and B. Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*. IEEE, 2019.

[42] J. Lei, L. Li, L. Zhou, Z. Gan, T. L. Berg, M. Bansal, and J. Liu. Less is more: Clipbert for video-and-language learning via sparse sampling. In *Proc. IEEE/CVF Conf. Comput. Vision Pattern Recognition (CVPR)*, 2021.

[43] W. Li, X. Zhang, Y. Wang, Z. Yan, and R. Peng. Graph2seq: Fusion embedding learning for knowledge graph completion. *IEEE Access*, 7, 2019.

[44] X. Li, Y. Qu, and H. Yin. PalmTree: Learning an assembly language model for instruction embedding. In *Proc. ACM SIGSAC Conf. Comput. and Commun. Security (CCS)*, 2021.

[45] X. Li, B. Trabucco, D. H. Park, M. Luo, S. Shen, T. Darrell, and Y. Gao. Discovering non-monotonic autoregressive orderings with variational inference. In *Int. Conf. Learning Representations (ICLR)*, 2021.

[46] X. Li, X. Yin, C. Li, P. Zhang, X. Hu, L. Zhang, L. Wang, H. Hu, L. Dong, F. Wei, Y. Choi, and J. Gao. Oscar: Object-semantics aligned pre-training for vision-language tasks. In *Proc. European Conf. Comput. Vision (ECCV)*, volume 12375 of *Lecture Notes in Comput. Science*. Springer, 2020.

[47] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. Graph matching networks for learning the similarity of graph

structured objects. In *Proc. Int. Conf. Machine Learning (ICML)*, 2019.

[48] Y. Liao, X. Jiang, and Q. Liu. Probabilistically masked language model capable of autoregressive generation in arbitrary word order. In *Proc. Annu. Meeting Association for Computational Linguistics (ACL)*. ACL, 2020.

[49] C. Lin and F. J. Och. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proc. Annu. Meeting Association for Computational Linguistics (ACL)*. ACL, 2004.

[50] C. Lin and F. J. Och. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *Int. Conf. Computational Linguistics (COLING)*, 2004.

[51] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou. αdiff: cross-version binary code similarity detection with DNN. In *IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*. ACM, 2018.

[52] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti. RE-mind: a first look inside the mind of a reverse engineer. In *USENIX Security Symposium, (USENIX)*. USENIX Association, 2022.

[53] L. Massarelli, G. A. D. 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)*, volume 11543. Springer, 2019.

[54] G. Monea, A. Joulin, and E. Grave. Pass: Parallel speculative sampling. *workshop Efficient Natural Language and Speech Processing (ENLSP), (NeurIPS)*, abs/2311.13581, 2023.

[55] V. Nitin, A. Saieva, B. Ray, and G. Kaiser. DIRECT: A transformer-based model for decompiled variable name recovery. *Workshop Natural Language Processing for Programming (NLP4Prog)*, 2021.

[56] J. Patrick-Evans, L. Cavallaro, and J. Kinder. Probabilistic naming of functions in stripped binaries. In *Proc. 35th Annu. Comput. Security Applications Conf. (ACSAC)*. ACM, 2020.

[57] J. Patrick-Evans, M. Dannehl, and J. Kinder. Supplementary material for XFL. https://github.com/lmu-plai/xfl, 2023.

[58] J. Patrick-Evans, M. Dannehl, and J. Kinder. XFL: Naming functions in binaries with extreme multi-label learning. In *Proc. IEEE Symp. Security and Privacy (S&P)*. IEEE, 2023.

[59] 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 SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE)*, 2021.

[60] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2021.

[61] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T. Liu. How could neural networks understand programs? In *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2021.

[62] W. Qi, Y. Yan, Y. Gong, D. Liu, N. Duan, J. Chen, R. Zhang, and M. Zhou. ProphetNet: Predicting future n-gram for sequence-to-sequence pre-training. *arXiv preprint arXiv:2001.04063*, 2020.

[63] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision. In *Proc. Inf. Conf. Machine Learning (ICML)*, volume 139 of *PMLR*, 2021.

[64] A. Santilli, S. Severino, E. Postolache, V. Maiorca, M. Mancusi, R. Marin, and E. Rodolà. Accelerating transformer inference for translation via parallel decoding. In *Proc. Annu. Meeting Association for Computational Linguistics (ACL)*. ACL, 2023.

[65] O. Sener and V. Koltun. Multi-task learning as multi-objective optimization. *Annu. Conf. Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[66] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *Proc. IEEE Symp. Security and Privacy (S&P)*, 2016.

[67] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, 2012.

[68] K. Song, X. Tan, T. Qin, J. Lu, and T. Liu. MASS: masked sequence to sequence pre-training for language generation. In *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2019.

[69] M. Stern, W. Chan, J. Kiros, and J. Uszkoreit. Insertion transformer: Flexible sequence generation via insertion operations. In *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2019.

[70] R. Sun, S. Guo, J. Guo, W. Li, X. Zhang, X. Guo, and Z. Pan. GraphMoCo: A graph momentum contrast model for large-scale binary function representation learning. *NeuroComputing*, 575, 2024.

[71] R. Team. *Radare2 Book*. GitHub, 2017.

[72] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Annu. Conf. Neural Information Processing Systems (NeurIPS)*, 2017.

[73] J. Venugopalan, L. Tong, H. R. Hassanzadeh, and M. D. Wang. Multimodal deep learning models for early detection of alzheimer's disease stage. *Scientific reports*, 11(1), 2021.

[74] D. Votipka, S. M. Rabin, K. K. Micinski, J. S. Foster, and M. L. Mazurek. An observational investigation of reverse engineers' processes. In *USENIX Security Symposium (USENIX)*. USENIX Association, 2020.

[75] H. Wang, Z. Gao, C. Zhang, Z. Sha, M. Sun, Y. Zhou, W. Zhu, W. Sun, H. Qiu, and X. Xiao. CLAP: learning transferable binary code representations with natural language supervision. In *ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2024.

[76] 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 Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2022.

[77] J. Wang, Z. Yang, X. Hu, L. Li, K. Lin, Z. Gan, Z. Liu, C. Liu, and L. Wang. GIT: A generative image-to-text transformer for vision and language. *Trans. Mach. Learn. Res.*, 2022.

[78] P. Wang, A. Yang, R. Men, J. Lin, S. Bai, Z. Li, J. Ma, C. Zhou, J. Zhou, and H. Yang. OFA: unifying architectures, tasks, and modalities through a simple sequence-to-sequence learning framework. In *Proc. Int. Conf. Machine Learning (ICML)*, volume 162. PMLR, 2022.

[79] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 2021.

[80] H. Wei, R. Xie, H. Cheng, L. Feng, B. An, and Y. Li. Mitigating neural network overconfidence with logit normalization. In *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2022.

[81] A. Wettig, T. Gao, Z. Zhong, and D. Chen. Should you mask 15% in masked language modeling? In *Proc. of the Conf. of the European Chapter of the ACL (EACL)*, pages 2985–3000. Association for Computational Linguistics, 2023.

[82] Q. Xie, M.-T. Luong, E. Hovy, and Q. V. Le. Self-training with noisy student improves imagenet classification. In *Proc. IEEE/CVF Conf. Comput. Vision Pattern Recognition (CVPR)*, 2020.

[83] 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 Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2023.

[84] X. Xu, S. Feng, Y. Ye, G. Shen, Z. Su, S. Cheng, G. Tao, Q. Shi, Z. Zhang, and X. Zhang. Improving binary code similarity transformer models by semantics-driven instruction deemphasis. In *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2023.

[85] 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 *Proc. ACM SIGSAC Conf. Comput. and Comm. Security (CCS)*. ACM, 2017.

[86] 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. In *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 2023.

[87] W. Ye, Y. Ma, X. Cao, and K. Tang. Mitigating transformer overconfidence via lipschitz regularization. In *Uncertainty in Artificial Intelligence (UAI)*. PMLR, 2023.

[88] J. Yu, Z. Wang, V. Vasudevan, L. Yeung, M. Seyedhosseini, and Y. Wu. CoCa: Contrastive captioners are image-text foundation models. *Trans. Mach. Learn. Res.*, 2022.

[89] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Conf. Artificial Intelligence (AAAI)*. AAAI Press, 2020.

[90] L. Yuan, D. Chen, Y.-L. Chen, N. Codella, X. Dai, J. Gao, H. Hu, X. Huang, B. Li, C. Li, et al. Florence: A new foundation model for computing vision. *arXiv preprint arXiv:2111.11432*, 2021.

[91] P. Zhang, X. Li, X. Hu, J. Yang, L. Zhang, L. Wang, Y. Choi, and J. Gao. VinVL: Revisiting visual representations in vision-language models. In *Proc. IEEE/CVF Conf. Comput. Vision Pattern Recognition (CVPR)*, 2021.

[92] R. Zhang, Z. Zeng, Z. Guo, and Y. Li. Can language understand depth? In *ACM Int. Conf. on Multimedia (MM)*. ACM, 2022.

[93] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi. BERTScore: Evaluating text generation with BERT. In *Int. Conf. Learning Representations (ICLR)*, 2020.

[94] X. Zhang, W. Sun, J. Pang, F. Liu, and Z. Ma. Similarity metric method for binary basic blocks of cross-instruction set architecture. In *Proc. Workshop Binary Analysis Research (BAR)*, 2020.

[95] Y. Zhang, C. Huang, Y. Zhang, K. Cao, S. T. Andersen, H. Shao, K. Leach, and Y. Huang. Pre-training representations of binary code using contrastive learning. *arXiv preprint arXiv:2210.05102*, 2024.

[96] F. Zhao, C. Zhang, and B. Geng. Deep multimodal data fusion. *ACM Comput. Surv.*, 56(9), 2024.

[97] 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.

A Automatic Function Names

To accurately simulate a real-world scenario of using name prediction, e.g., within an advanced disassembler, we follow Patrick-Evans et al. [58] by treating twenty functions as automatically inferred in each metric: `init`, `fini`, `csu_init`, `csu_fini`, `start`, `libc_csu_init`, `libc_csu_fini`, `libc_start`, `deregister_tm_clones`, `register_tm_clones`, `rtld_init`, `main`, `do_global_dtors_aux`, `frame_dummy`, `frame_dummy_init_array_entry`, `do_global_dtors_aux_fini_array_entry` `init_array_end`, `init_array_start`, `start_main`, `libc_start_main`. This list matches those functions automatically labeled by IDA Pro.

B Case Study

```c
int execute(char *program, char *action,
            char *stardata) {
  char *path; int result;
  path = malloc(strlen(SHELL)+strlen(SCRIPTS_DIR)+
               strlen(program)+strlen(action)+
               strlen(stardata)+4);
  if (!path) { return 1; }
  sprintf(path, "%s %s%s %s %s", SHELL,
       SCRIPTS_DIR, program, action, stardata);
  result = system(path); free(path);
  return result;
}
```

Listing 1: Case ❶ source code.

```c
void evalbackcmd(union node *n,
                 struct backcmd *result) {
  int pip[2]; struct job *jp;
  result->fd = -1; result->buf = NULL;
  result->nleft = 0; result->jp = NULL;
  if (n == NULL) { goto out; }
  if (pipe(pip)<0){sh_error("Pipe call failed");}
  jp = makejob(n, 1);
  if (forkshell(jp, n, FORK_NOJOB)==0) {
    FORCEINTON; close(pip[0]);
    if (pip[1] != 1) { dup2(pip[1], 1);
                       close(pip[1]);  }
    ifsfree(); evaltreenr(n, EV_EXIT);
  }
  close(pip[1]);
  result->fd = pip[0]; result->jp = jp;
  out:
    TRACE(("evalbackcmd done: fd =%d buf=0x %x
    nleft=%d jp=0x%x\n", result->fd, result->buf,
    result->nleft, result->jp));
}
```

Listing 2: Case ❷ source code.

```c
GKeyFile *fsa_get_config(void) {
  GKeyFile *config_file = g_key_file_new();
  GError *err = NULL;
  const char *filename = fs_get_config_file();
  int rv; int flags = G_KEY_FILE_NONE;
  rv = g_key_file_load_from_file(config_file,
    filename, flags, &err);
  if (!rv || err != NULL) {
    if (err->code != G_FILE_ERROR_NOENT && err->
    code != G_FILE_ERROR_EXIST && err->code !=
    G_FILE_ERROR_ISDIR)
    {fsa_error(LOG_ERR, "error reading %s: %s(%d)"
    , filename, err->message, err->code );}
    g_error_free(err);
    g_key_file_free(config_file);
    errno = ADM_ERR_GENERIC; return NULL;
  }
  errno = 0; return config_file;
}
```

Listing 3: Case ❸ source code.

```c
struct tws* dtwstime() { // from dtime.c
    long clock; (void) time( &clock );
    return dlocaltime( &clock );
}
struct tws { // from tws.h
    int tw_sec; int tw_min; int tw_hour;
    int tw_mday; int tw_mon; int tw_year; ... };
```

Listing 4: Case ❹ source code.

```c
void removeFromEdited(INT base, int size) {
  typePage *p, *q = NULL;
  for (p = edited; p; p ? (q = p, p = p->next) : (
    q = NULL, p = edited)) {
    if (base + size <= p->base) break;
    if (base <= p->base) {
      if (p->base + p->size <= base + size) {
        if (q) {q->next = p->next};
        else {edited = p->next};
        freePage(p); p = q;
      } else { p->size -= base+size - p->base;
               memmove(p->vals, p->vals + base
               + size - p->base, p->size);
               p->base = base + size; }
    } else if (p->base + p->size <= base + size) {
      if (base < p->base + p->size)
        {p->size -= p->base + p->size - base};
    } else { q = newPage(base + size, p->base
             + p->size - base - size);
             memcpy(q->vals, p->vals + base
             + size - p->base, q->size);
             q->next = p->next; p->next = q;
             p->size -= p->base + p->size - base;
             break; }
  }
  updatelastEditedLoc();
}
```

Listing 5: Case ❺ source code.

```c
int taskpanel_cancel_clicked_cbk(GtkButton *btn,
                                 gpointer data) {
  log_fct_enter();
  ui_taskpanel_update(current_task);
  log_fct_exit(); return FALSE; }
```

Listing 6: Case ❻ source code.