You Can't Judge a Binary by Its Header: Data-Code Separation for Non-Standard ARM Binaries using Pseudo Labels (Supplementary Materials)

Hadjer Benkraouda
University of Illinois at Urbana-Champaign
hadjerb2@illinois.edu

Nirav Diwan
University of Illinois at Urbana-Champaign
ndiwan2@illinois.edu

Gang Wang
University of Illinois at Urbana-Champaign
gangw@illinois.edu

1. BERT Implementation Details

We offer an alternative implementation of the proposed framework in our main paper. This alternative implementation uses a large language model (i.e., BERT [1]) to handle both instruction embedding and classification. While this variant can also accurately perform data-code separation, we did not choose this implementation as the default for `Loadstar` for its inference overhead. In the following, we describe the BERT implementation in detail.

Embedding Model. We pre-train a BERT model to act as the embedding model (❶ in Figure 1). The embedding model seeks to map an assembly instruction to a fixed-size vector. Following recent literature [2], the pre-training only uses the Masked Language Modelling (MLM) as the self-supervision task to effectively train on large datasets.

The pre-training of the embedding model uses the standard binaries (dataset S). We pre-process the data to allow the model to learn the contextual representations of tokens. Specifically, a context window of size 2 is used, which involves augmenting each instruction with two neighboring instructions on both sides. For instance, consider the target instruction "`movweq r0, 0x5f14`" below (highlighted):

```
1    movweq r0, 0x5f0a
2    addeq r8, r0, r3, ror r4
3    movweq r0, 0x5f14
4    addeq r8, r0, r7, ror 8
5    movweq r0, 0x5f13
```

For a context window size of 2, we include the two instructions before the target instruction and the two instructions after it to create a sequence. [CLS] and [SEP] are special tokens that indicate the beginning and end of the sentence. The pre-processed instruction would look as follows: "*[CLS] movweq r0, 0x5f14 movweq r0, 0x5f0a addeq r8, r0, r3, ror r4 addeq r8, r0, r7, ror 8 movweq r0, 0x5f13 [SEP]*"

Next, the samples are tokenized using the Wordpiece tokenizer [1]. We set the vocabulary size to 2048 for the tokenizer. The tokenizer works by making common words or subwords into single tokens while breaking less common words into smaller parts. This helps to improve its ability to generalize and handle unknown words during testing time. On average, each sample is tokenized to 52 tokens.

Next, we train the BERT model for Masked Language Modelling (MLM). This task involves masking about 15% of the tokens and training the model to correctly predict the masked tokens. The masking process works as follows. For each input sequence, 15% of the WordPiece tokens are selected at random to be masked. Of the selected tokens, 80% are replaced with the [MASK] token, 10% are replaced with a random token, and 10% are left unchanged. The model is then tasked to predict the missing [MASK] tokens. The pre-training of the model is formalized through the optimization of a loss function:

$$\mathcal{L}_{\text{pre}} = -\sum_{i \in M} \log(p_{\theta_{EM}}(x_i \mid x_{\setminus i})) \tag{1}$$

where L_{pre} is the loss for a single training example, M is the set of indices of the masked tokens in the input sequence, x_i is the actual token at position i, $p_\theta(x_i \mid x_{\setminus i})$ represents the probability predicted by the model for the actual token x_i, given the other unmasked tokens $x_{\setminus i}$, and θ_{EM} denotes the parameters of the model.

The objective during training is to minimize this loss across all training examples. For the training, we use the default hyperparameters from BERT. The model has 12 self-attention heads in the multi-headed attention mechanism. The size of the resultant vector of an instruction is 768.

Classification Model. For the classification model (❷ in Figure 1), we fine-tune the BERT model for the data-code separation task. This is done using the *labeled* dataset of standard binaries (S). Due to the high training overhead, we only use a random sample of 100,000 instructions from S to perform fine-tuning. We keep the balance between "data" and "code" instructions in the sample set from the full dataset.

For this classifier, we add three fully connected layers to the pre-trained embedding model. The final layer contains two units, corresponding to the two classes ("code" or "data"). A softmax layer calculates the probability that the input is either code or data, based on the model's output. The model is initialized with the training weights θ_{EM} from the previous phase. The fine-tuning process is governed by the following

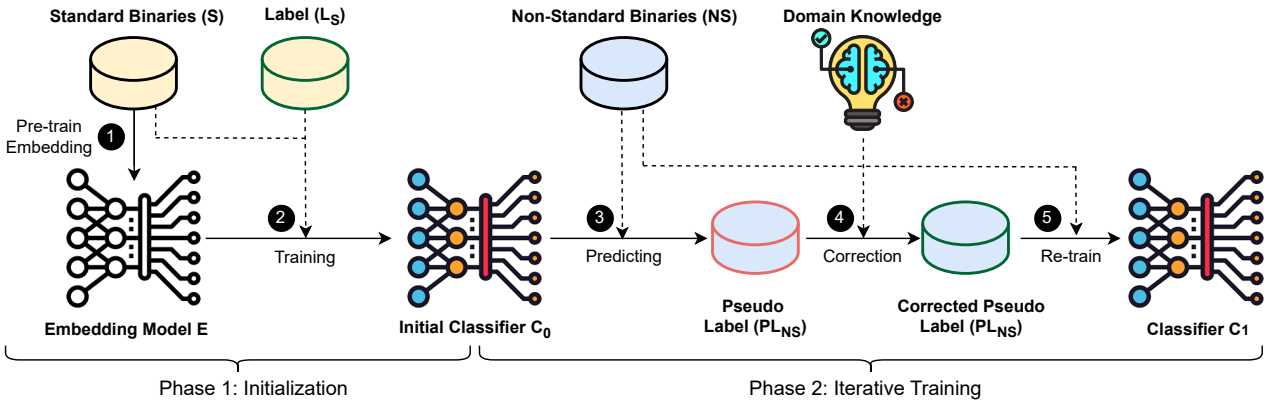

Figure 1: Overview of `Loadstar`.

cross-entropy loss function:

$$\mathcal{L}_{\text{ft}} = -\sum_{j} \log p(y_j \mid x_j; \theta_{CM}) \tag{2}$$

where x_j is the data instance, y_j is the corresponding true label, and θ_{CM} represents the model parameters updated during this supervised fine-tuning stage.

Following the seminal BERT paper [1], we train the classification model for 4 epochs using the AdamW optimizer with a learning rate of $2e-6$, an epsilon value of $1e-8$, and β_1 and β_2 values of 0.9 and 0.999, respectively. We include a weight decay of 0.01. The batch size is set to 16, and the model is trained on a GPU.

Iterative Training. For the BERT model, the rest of the steps in Figure 1 are the same with what is described in the main paper. These steps (❸–❺) include pseudo-label generation and correction, and model retraining. For retraining, again due to the high overhead, we use a sampled set of 100,000 instructions from non-standard binaries (with curated pseudo labels).

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *NAACL*, 2019.

[2] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *ICLR*, 2020.