BinBert: Binary Code Understanding With a Fine-Tunable and Execution-Aware Transformer

Fiorella Artuso, Marco Mormando, Giuseppe Antonio Di Luna, and Leonardo Querzoni

Abstract—a recent trend in binary code analysis promotes the use of neural solutions based on instruction embedding models. An instruction embedding model is a neural network that transforms assembly instructions into embedding vectors. If the embedding network is able to process sequences of assembly instructions transforming them into a sequence of embedding vectors, then the network effectively represents an assembly code model. In this paper, we present BinBert, a novel assembly code model. BinBert is built on a transformer pre-trained on a huge dataset of both assembly instruction sequences and symbolic execution information. BinBert can be applied to assembly instructions sequences and it is fine-tunable, i.e., it can be re-trained as part of a neural architecture on task-specific data. Through fine-tuning, BinBert learns how to apply the general knowledge acquired with pre-training to the specific task. We evaluated BinBert on a multi-task benchmark that we specifically designed to test the understanding of assembly code. The benchmark is composed of several tasks, some taken from the literature, and a few novel tasks that we designed, with a mix of intrinsic and downstream tasks. Our results show that BinBert outperforms state-of-the-art models for binary instruction embedding, raising the bar for binary code understanding.

Index Terms—Binary analysis, machine learning.

I. INTRODUCTION

A GROWING body of literature has demonstrated that Deep Neural Networks (DNNs) can effectively address various binary analysis tasks. DNNs today show state of the art performances for binary similarity [1], [2], [3], compiler provenance [4], [5], [6], function boundaries detection [7], de-compiling [8], automatic function naming [9], [10] and others.

DNN designers must decide how to feed binary code to their models. One possibility is to use manually-identified features. This approach requires a domain expert which identifies features of interest forecasting their helpfulness in solving the task at hand. This approach is known to produce problem-specific features and injects a human bias inside the system. Recent solutions automatically transform binary code into a representation usable by the neural network layers.

A common technique is to transform assembly instructions into representational embeddings vectors, similar to what has been done in the Natural Language Processing (NLP) field with the word embedding revolution [11]. Several works [1], [2], [12], [13] proposed refined techniques to transform a single instruction into a vector of real numbers while capturing its semantic (e.g., all vectors of arithmetic instructions are clustered in the vector space). By using this approach, sequences of instructions are transformed into sequences of fixed-size vectors that can be fed into standard DNNs.

A common weakness of all these approaches is the lack of context: an instruction is always represented by the same vector, irrespectively of where it appears. However, the semantics of a single assembly instruction is strongly limited (more than a word in natural language), and non-trivial concepts in assembly code are almost always encoded by a sequence of instructions (e.g., loops, swap of variables in memory, calling conventions, etc). Complex semantics, that span sequences of several assembly instructions, are hardly representable if embeddings of instructions are created in isolation; they have to be learned by the neural architecture using the embeddings.

Recent works [14], [15] overcome this limitation by using a transformer [16] based architecture that operates on sequences of assembly instructions. This enables them to embed entire assembly functions taking into consideration the instructions’ context. These systems have not been proposed and tested as instruction embedding techniques but as solutions to specific problems (Trex [14] for binary similarity and the Stateformer [15] for type inference).

A. Execution-Aware Binary Code Interpretation

Code serves as a form of communication between humans and machines, possessing a dual nature. One aspect represents the syntactic and semantic meaning that can be inferred from its static form, while the other aspect lies in its ability to be executed. The full understanding and appreciation of an Instruction Set Architecture (ISA) can only be achieved through the execution of code (e.g., the dependencies introduced by RFLAGS in X64).

Additionally, sequences of instructions that have the same meaning but a different syntax can be easily identified when they are executed.

Surprisingly, almost all embedding techniques we are aware of, only consider the static aspect of binary code. Notable exceptions are the aforementioned [14], [15] in which the execution is embedded by training the models on the assembly instructions of a function and on the values of CPU registers obtained by a concrete execution with random inputs. One could argue that
such an execution approach is prone to the noise and the limited significance of a random execution.

B. Expressive Power and Fine-Tunable Models

Oddly, the existing instructions and function embedding models are proposed and tested on a single problem ([14], [15], [1], [12]). This is limiting as the expressive power of an embedding model can only be assessed when the model is tested on different tasks. With the current body of knowledge, there is uncertainty on whether and how the proposed embedding models generalize to different tasks or not.

The only exception is Palmtree [13], which proposes an assembly instruction model tested on a few tasks. However, a glaring limitation of [13] is that their embedding model is frozen and not fine-tuned.

We argue that an assembly model has to be tested using the fine-tuning paradigm. That is, the model is first pre-trained on a large corpus of assembly code using several tasks. During pre-training, the model learns a general semantics of assembly sequences that is context and execution aware. Then, the pre-trained model is used as part of a DNN that solves a specific downstream task (e.g., compiler provenance, function similarity, and others). The DNN, including the assembly model, is retrained end-to-end on a small amount of problem-specific data during the fine-tuning process. This paradigm is state-of-the-art for NLP and works well also if the fine-tuning dataset is small. This is especially useful for binary analysis tasks where creating a labeled dataset requires expensive manual effort.

C. Our Proposal: BinBert

In this paper we introduce BinBert, a fine-tunable assembly code model based on a transformer encoder that is execution-aware. To inject execution awareness into our model, our idea is to symbolically execute snippets of assembly code. Specifically, we use a symbolic execution engine that transforms sequences of assembly instructions connected by a data-dependency relationship (the strands introduced in [17]) into sets of semantically equivalent symbolic expressions. These expressions are a functional representation of the input-output relationship of the strand. We designed a novel pre-training process that forces BinBert to learn the correct matching between an assembly sequence and an equivalent symbolic expression and to translate assembly code into symbolic expressions and vice-versa. Our intuition is that symbolic expressions are more useful than using randomised concrete executions as they do not suffer from the same level of noise.

We train BinBert on a new large dataset\(^1\) of assembly sequences and symbolic expressions derived from symbolic execution, obtaining a general-purpose assembly code model. Our model is able to create representative embedding of single instructions, as well as to generate representative embeddings of sequences of instructions that could be either snippet of assembly code or entire functions. We remark that symbolic execution is needed only in the pre-training phase; no code execution is required while using the model for inference tasks.

We tested BinBert on a multi-task benchmark for binary code understanding that we built. Tasks in the benchmark range from intrinsic ones, aimed at evaluating how the pre-trained BinBert captures the semantic of instructions and sequences, to extrinsic downstream tasks, in which we fine-tune BinBert for problems on assembly sequences and binary functions. In all our experiments BinBert raises the performance bar outperforming the current state-of-the-art (including PalmTree [13]) and specific solutions created for the binary similarity problem.

In summary, this paper provides the following contributions:

- a novel training task that makes the training of an assembly code model execution-aware by using symbolic expressions derived from the symbolic executions of assembly snippets;
- BinBert, a pre-trained execution-aware transformer model for X64, that can be plugged into DNNs for binary analysis. The model has been pre-trained on a 26 GByte dataset. We release the model, the code used to train it as well as the dataset;
- the first multi-task benchmark designed to test the binary code understanding of assembly models. The benchmark is composed of well-known tasks selected from the literature for their relevance, and two novel tasks (strand recovery and execution) for the semantic understanding of assembly sequences;
- an in-depth performance evaluation of BinBert based on our benchmark that shows how execution awareness improves the performance of an assembly model. To the best of our knowledge, we are the first to thoroughly test the impact of the fine-tuning paradigm on assembly representation learning. We show that, as already shown in the NLP field, the pre-training/fine-tuning approach has a positive impact on all downstream tasks. As a consequence, BinBert outperforms the current state-of-the-art instruction embedding techniques.

II. BACKGROUND

In this section, we introduce the general theoretical concepts behind the instruction embedding techniques and focus on the current state of the art. Afterwards, we detail weak points and gaps in current solutions, discussing how these influenced our proposal.

A. Instruction Embedding Models

An instruction embedding model takes as input an assembly instruction \(i\) from a vocabulary \(V\) of size \(d\) and it returns a vector of real numbers \(e(i) = \hat{v} \in \mathbb{R}^n\), \(n\) is the embedding size (typically \(n \in \{128, 1024\}\)). The vector \(\hat{v}\) is a dense representation of the instruction \(i\).

In the simplest embedding scheme a random matrix \(M\) of size \(\mathbb{R}^{d \times n}\) is created, each instruction is mapped univocally to a row of \(M\). A sequence of instructions \(I = [i_0, i_1, \ldots, i_m]\) is converted into a sequence of vectors \(e(I) = [\hat{v_0}, \hat{v_1}, \ldots, \hat{v_m}]\) using a lookup mechanism. This sequence is fed into the task-specific

\(^1\) Size-wise our dataset is larger than the original dataset used to train Bert [18].
DNN \( A \). The matrix \( M \) is usually \textit{trainable}: its elements are trainable weights and are modified during the training of \( A \).

The groundbreaking idea of the embedding models is to generate the embedding matrix \( M \) with a neural network \( \text{Emb} \), formally speaking \( M = \text{Emb}(A) \). The network \( \text{Emb} \) is trained in an unsupervised way on a corpus \( C \) of data. This corpora \( C \) is composed of sequences of assembly instructions extracted from selected binaries. Usually, the \textit{distributed representation learning tasks} used by instruction embedding models are, apart from minimal modifications, the ones used in NLP by solutions such as \text{word2vec} [11], GloVe [19], \text{fastText} [20], \text{pv-dm} [21].

The common goal is to train \( \text{Emb} \) to produce an embedding vector that contains enough information to predict a masked instruction from its context in \( C \). Most of the novelty of the instruction embedding system is in the preprocessing of instructions and in the definition of the assembly sequences composing \( C \).

1) \textit{Preprocessing of Assembly Instructions}: Assembly language and natural language are distinguished by the wide difference between the vocabulary size \( \hat{d} \). A natural language is usually composed of hundred thousands different words, while the number of possible distinct assembly instructions is much more. Consider the X64 ISA, a mov instruction can use 64 bits to express immediates, offsets, and memory addresses, thus there can be \( 2^{64} \) different instructions that just move a value in a certain register. This makes raw assembly instructions impractical: a large vocabulary is discouraged [22] as it worsens the problem of out-of-vocabulary word (OOW) [23].

Moreover, the exact value of an immediate is largely useless in a static analysis setting (e.g. a memory address of an unknown memory layout) [1]. To ameliorate this problem, a lot of effort has been devoted to instructions preprocessing [1], [2], [13], [14], [24], [25]. The standard of the field is to substitute all memory addresses and immediates above a certain threshold value with special symbols (e.g. \text{IMM}). Another design choice is whether to consider the entire assembly instructions as a token (used in [1]), to split the assembly instructions into several tokens by separating opcodes and operands (used in [25]) or to use a more fine-grained split strategy [13]. Interestingly, no one used automatic tokenization such as WordPiece [26] that are standard in NLP.

2) \textit{Extraction of Assembly Sequences}: A key point is how to extract the sequences from the binary, as this defines the context in which an instruction appears. The context for instruction \( i_x \) is composed by \( k \) instructions appearing before/after \( i_x \) in \( C \). Used extraction strategies are:

- \textit{Linearized Control Flow Graph (CFG)}: In this case each sequence is a linearization of a CFG (commonly the one provided by a disassembler) [1]. Blocks of the CFG that are not logically related could be sequentially placed in the linearization, and thus an instruction will see a noisy context. An example is the linearization of the CFG in Fig. 1 induced by instructions numbers: the context of instruction \( i_8 \) contains instructions \( i_9, i_{10} \) that are not causally related. This injects noise into the learning process.

- \textit{Control Flow Graph (CFG)/Interprocedural Control Flow Graph (ICFG)}: The sequences are extracted from the recovered CFG/ICFG. This is done either by using a random walk strategy ([25]), or by taking as a sequence a single block [2]. The idea is to have a sequence that respects the logical control flow of the examined program, removing the source of noise highlighted in the previous strategy. We argue that this technique does not completely remove the presence of extraneous instructions in the context. Take the sequence of instructions \( i_5, i_6, i_7, i_8 \) in Fig. 1, the context of instruction \( i_7 \) contains \( i_6 \) and \( i_8 \) that are not causally related.

3) \textit{Transformer Based Solutions}: Recent instructions embedding models are based on transformers. In this case the embedding network is not a single matrix, but a complex DNN that is able to transform a sequence of instructions into vectors. The considerations above also applies to these solutions, they have to decide how to extract assembly sequences and how to preprocess such sequences.

\textbf{a) PalmTree}: PalmTree [13] is a transformer-based instruction embedding model that has shown state-of-the-art performances beating all the other embedding models on several tasks. Instructions are divided into tokens using a fine-grained strategy with manually made regexs. The model is trained on pairs of instructions taken from the corpora \( C \). PalmTree uses the standard Masked Language Modeling (MLM) of Bert and two novel tasks: the Context Window Prediction task (CWP) in which the network has to recognise if a pair of instructions is taken from the same context or not, and the Def-Use Prediction

\[ i_0: \text{mov eax, 5} \quad s_1 \quad s_2 \]
\[ i_1: \text{mov ebx, [MEM]} \quad s_1 \]
\[ i_2: \text{sub ecx, eax} \quad s_2 \]
\[ i_3: \text{cmp ebx, eax} \quad s_1 \]
\[ i_4: \text{jin BLK1} \quad s_1 \]
\[ i_5: \text{xor ecx, ecx} \quad s_3 \]
\[ i_6: \text{add ebx, 1} \quad s_4 \]
\[ i_7: \text{add ecx, 2} \quad s_3 \]
\[ i_8: \text{mov [ebx], eax} \quad s_4 \]
\[ i_9: \text{mov ecx, 1} \quad s_6 \]
\[ i_{10}: \text{mov [ebx], eax} \quad s_6 \]
\[ i_{11}: \text{ret} \quad s_7 \]

Fig. 1. CFG of an imaginary function in x64 assembly. For each instruction we indicate with \( s \) the block level strand to which it belongs.

\[^2\text{A Control Flow Graph (CFG) represents the flow of control in a function and shows the order in which instructions are executed. The CFG is composed of a set of basic blocks (where each block represents a sequence of instructions) connected by edges (where each edge represents a conditional or an unconditional branch.)}\]

\[^3\text{An Interprocedural Control Flow Graph (ICFG) is an extension of the traditional CFG since it covers multiple functions within a program}\]
task (DUP) in which the network has to recognize if there is a data dependency between instructions. Once trained, the model is used as an instruction embedding model: a sequence of instructions is embedded by applying separately the PalmTree model to each instruction.

b) Trex and Stateformer: Trex and Stateformer are two solutions that use micro-execution traces and transformer architecture. Both utilize similar pre-training strategies and have similar architectures but they are used to solve different tasks: Trex is used for function similarity, while Stateformer is used for variables type recovery. The transformer in these solutions is fed with five sequences: the assembly code sequence, the micro-trace value sequence and additional information sequences that specify the order of instructions, architecture, and the position of opcodes. Regarding assembly sequence, the authors treat all symbols as tokens, including punctuation. To generate embeddings for the micro-trace values, Trex uses a Bi-LSTM, while Stateformer uses a Neural Arithmetic Unit. In terms of pre-training, Trex focuses on predicting masked codes and values in micro-traces, while Stateformer is trained on predicting micro-trace values and whether a particular instruction is executed in a trace. Differently from PalmTree, instruction embeddings are not produced in isolation.

III. THE BINBERT SOLUTION

In this section we describe BinBert. We first give an overview of the system, briefly describing the transformer architecture [16]. We then give the details of the innovative aspects of BinBert.

A. Overview

The neural architecture of BinBert is the standard transformer encoder [16] used by Bert [18]. A transformer encoder processes sequential data using an attention mechanism, which allows for both the creation of more informative embeddings (by focusing only on relevant parts of the sequence) and good performances (the attention mechanism is implemented using matrix multiplication that is highly parallelizable on GPUs). More specifically, a transformer encoder is composed of $N$ identical layers stacked one on top of the other, where each layer consists of two sublayers: a multi-head self-attention mechanism and a fully connected feed-forward network. Practically speaking, a sequence of $n$ tokens is transformed into a sequence of $n$ latent vectors (one for each token) with a mechanism that we will explain in Section III-D; this sequence is fed into the initial layer of the encoder. Each other layer takes as input the hidden state token vectors returned by the previous layer. The output of the encoder is a sequence of $n + 1$ embedding vectors: one for each input token and a special embedding for the entire sequence (the [CLS] vector described in Section III-D).

To avoid the vocabulary inflation generated by the use of raw assembly instructions, BinBert substitutes memory addresses

\[ \text{RD 2: Design an embedding model that can be trained end-to-end on a specific task, transferring the general knowledge learned during pre-training. The model has to be evaluated on a multi-task benchmark designed to thoroughly test the syntactic and semantic understanding of the assembly language.} \]

\[ \text{RD 1: Design an embedding model that takes into account the execution of code using symbolic execution. The impact of execution-related information has to be quantified with a specific ablation study.} \]

\[ ^4 \text{[13] explicitly states that the downstream tasks have been evaluated without fine-tuning since these tasks were implemented in Tensorflow 3 while their model was implemented in pyTorch.} \]
and immediates above a certain threshold with special symbols. Moreover, BinBert splits a single assembly instruction into several tokens using WordPiece [26].

In BinBert we decided to completely remove the noise given by instructions that are contextually related but have no logical relation (see Section II-A2) by extracting sequences representing strands [17]. Strands are sequences of causally related instructions computing the values of a certain variable. In this way, the context of an instruction never contains extraneous instructions introduced by compiler optimizations. We symbolically execute each strand to extract a set of symbolic expressions; these expressions will be used in our training tasks as a means to inject execution-related information into the pre-training.

During pre-training, BinBert learns the matching between symbolic expressions and strands (this is done using positive/negative pairs); at the same time, samples are partially masked forcing the model to guess the masked tokens by also learning a translation between symbolic expressions and assembly strands.

B. Instructions Preprocessing and Assembly Sequences Extraction

This Section will describe the instruction preprocessing rules that we have adopted together with the sequence of assembly instructions that we have used for training our model.

1) Instructions Preprocessing: We preprocess each assembly instruction substituting immediates above a threshold (5000 in our experiments) with the value \( \text{MEM} \) (the same is done for offsets and memory addresses). We use the special symbol \( \text{MEM} \) in case of jumps. We use a threshold-based approach as small immediate values are likely to carry informative content (comparison with small constants in branches and loops, PC/stack relative displacements that identify variables in memory). All immediates/offsets are converted to decimal format. For call instructions, we distinguish if the called function is user-defined or belongs to libc. For user-defined functions, we substitute the called address with \( \text{func} \) (our system is usable on stripped binaries). If it is a call to libc, we substitute the address with the function name (e.g., \( \text{call printf} \)), since external symbols cannot be stripped. Indirect calls are left untouched.

After preprocessing, each instruction is tokenized using WordPiece [26]. The latter uses a probabilistic approach to learn how to tokenize instructions in a way that minimizes the vocabulary size and the OOW problem. Contrarily to manually made regexes, WordPiece automatically learns how to split complex opcodes (as an example \( \text{cmovz} \) will be split in \( \text{cmov} \) and \( z \) helping the model in understanding the relationship between the cmovX family of X64). We use WordPiece also on symbolic expressions. This provides a uniform tokenization mechanism and vocabulary for the two distinct languages (asm/sym. expr.) used for BinBert.

2) Assembly Sequences: In BinBert we use the concept of strands to extract the sequences of assembly instructions on which our model is trained. This does not mean that BinBert cannot be fine-tuned and used on CFG blocks or entire functions as our experiments will show.

A strand, originally defined in [17], is a slice of a CFG block constituted by all the instructions that are connected by def-use dependences. More specifically, we consider as an output variable of a block a memory location or a register on which the last operation is a write or the check of a jump. Starting from this variable we construct a backward-slice of the block including all the instructions from which the value of such variables depends. To make the concept clear, consider the example in Fig. 1; the first block contains two strands S1 and S2: S1 is composed by instructions \( i_0, i_1, i_3, i_4 \) that influence the RFLAGS register later checked by \( i_4 \); strand S2 is composed by \( i_0, i_2 \) which define the value of variable \( ecx \). Other examples of strands are in the figure. We enrich the original definition of strand, by considering as a single strand all the instructions that prepare the input values for a call. In this case, the strand will be constituted by all the aforementioned instructions and the call instruction itself.

Therefore, we build our training corpora \( C \) by extracting the CFGs in binary, and then decomposing all the blocks in strands. The strands will be the basic sequences in \( C \).

This decomposition has several advantages: it completely removes the noise introduced by instructions that co-occur in the same context only for compiler optimization reasons; the model learns the entire “causal context” of an instruction so it is able to see long dependencies among instructions.

C. Symbolic Execution

In BinBert we employ symbolic execution to convert each strand into a representative symbolic expression. The symbolic execution engine operates on strands of assembly instructions before preprocessing (as it requires the actual values of immediates and memory locations). The engine is built on angr [27]. During the execution, we consider variables on which the strand’s first operation is a read as inputs, and variables on which the last operation is a write as outputs.

Each time the strand writes to a variable (either a memory location or a register), we express the written value using a symbolic expression. When a variable is read, it may either be an input (no one has written to it) or contain a symbolic value. If the variable is an input, we set its symbolic value to its address or register name (as example, for \( \text{mov eax, \{rbp+4\}} \) we have \( \text{eax} = *(\text{rbp} + 4) \), for \( \text{mov eax, ebx} \) we have \( \text{eax} = \text{ebx} \)).

The symbolic expression obtained with this process may have one of three possible forms:

- If the strand computes the value of a certain variable, the symbolic expression describes the value in the output variable as a function of the strand inputs. For example, in the first row of Table I, we

### Table I

| Strands | Symbolic Expressing |
|---------|---------------------|
| mov eax, esp | \( \text{mov} rax, \text{rax} + 8 \) |
| and eax, 0 | \( \text{and} rax, 0 \) |
| cmp byte ptr [rax], al | \( \text{cmp} [rax], \text{al} \) |
| mov eax, 0 | \( \text{mov} rax, 0 \) |
| mov ebx, [esp+4] | \( \text{mov} rax, \text{rbx} + 4 \) |
| mov ecx, [esp+4] | \( \text{mov} rax, \text{rcx} + 4 \) |
| mov edx, [esp+4] | \( \text{mov} rax, \text{rdx} + 4 \) |
| mov esi, [esp+4] | \( \text{mov} rax, \text{rsi} + 4 \) |
| mov edi, [esp+4] | \( \text{mov} rax, \text{rdi} + 4 \) |
| mov eax, [esp+4] | \( \text{mov} rax, \text{rax} + 4 \) |
| mov ecx, [esp+4] | \( \text{mov} rax, \text{rcx} + 4 \) |
| mov edx, [esp+4] | \( \text{mov} rax, \text{rdx} + 4 \) |
| mov esi, [esp+4] | \( \text{mov} rax, \text{rsi} + 4 \) |
| mov edi, [esp+4] | \( \text{mov} rax, \text{rdi} + 4 \) |

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
have the symbolic $\text{rcx} = -1$ add (0 Concat \text{rsi}[1:0]). This expression is for the output variable \text{ecx}; the and operation extracts the two least significant bits from \text{rsi}, the result is extended to 64 bits, and then decremented by the rep. The expression only contains the extended register of X64, which is a design choice that we discuss later in Section III-D.

- If the strand computes the predicate checked by a conditional branch, then our symbolic expression will be the comparison of the jump condition with a symbolic expression of the value used in the predicate. For instance, consider the second row of Table I; in this case, the symbolic expression is compared with 0 using the not equal (ne) predicate.

- Finally, if the strand computes the arguments used by a call instruction, our symbolic expression will be the call to the specific function (including the symbolic name if it is a libc call), where all the arguments are substituted by the symbolic expressions of their values. We extracted function arguments following the X64 calling convention.

An example of a call expression is in the third row of Table I. From a single strand we may obtain multiple symbolic expressions. For example, in the first row of Table I, the \text{rep} \text{stosb} instruction repeatedly places the content of al into the memory pointed by rdi for \text{ecx} times, decrementing \text{ecx} and incrementing rdi with each iteration. This means that the strand has three output variables (a memory location and two registers), and each one will have its symbolic expression. We will call this set of symbolic expressions the representative set of the strand.

a) Preprocessing and Tokenization of Symbolic Expressions: The symbolic expressions of each strand are preprocessed similarly to assembly. Large numerical constants are substituted with the special symbol IMM, for floating point numbers all the digits after two decimals are truncated. The symbolic expressions are tokenized using WordPiece. During the preprocessing phase, we substitute the name of all registers to their extended form (i.e., we use rax instead of eax). We do so to help the network in understanding the relationships between names used to address different parts of the same logical register; this step is not applied while preprocessing assembly instructions (i.e., we leave eax in the strand).

D. BinBert Input Representation and Pre-Training Tasks

BinBert is fed with pairs <strand, symbolic expression> and pre-trained on two tasks that we name Execution Language Modeling (ELM) and Strand-Symbolic Mapping (SSM).

1) Input Representation: A BinBert input consists of a tokenized strand-symbolic expression pair. Two special tokens are added to each sample: [SEP] is used to distinguish between assembly and symbolic expression, and [CLS] is prepended to all samples. The hidden state of the [CLS] token in the last hidden layer is typically used to obtain a latent vector representation of the entire sequence [18] (see Fig. 2).

We employ dynamic padding, padding shorter sequences with the special token [PAD], while truncating sequences longer than a threshold (we use 512 in our experiments). Before processing by the transformer architecture, each token is converted into a vector using the lookup mechanism described in Section II-A. Token embeddings are then summed with both position and language embeddings [18], [28]. Position embeddings enable the model to be aware of sequence order, while language embeddings help distinguish between assembly code and symbolic expressions (see Fig. 2).

2) Pre-Training Tasks: The first task is Execution Language Modeling (ELM). The objective of ELM is to mask a certain percentage $mp$ of tokens in the input pairs and have the network predict the original ones. As in Bert, the tokens to be predicted are either substituted with a special token [MASK], replaced with a random one, or left untouched with probabilities of 80-10-10, respectively. Fig. 2 shows an example in which the tokens rax, mov, and 3 are masked, and the network attempts to reconstruct them as output. Note that in order to reconstruct a token contained in a strand, the network must pay attention to both the strand and its corresponding symbolic expression (the same applies for a token in the symbolic expression). This means that to predict the value 3 for register rbx, the network is forced to understand the data flow in the strand, thus making the semantic of each strand instruction explicit. As in [18], a linear layer is added on top of the last hidden layer, specifically to the hidden states of the masked tokens; this layer will guess the masked tokens.

Mathematically speaking, we have a dataset $D$ of <strand, symbolic expression> pairs $I = \{t_1, \ldots, t_n\} \in D$, where each token $t_i$ belongs to a vocabulary $V \in \mathbb{R}^d$. We feed each pair to BinBert to obtain context-aware hidden vectors $H = \text{Binbert}(I) = [\vec{h}_1, \ldots, \vec{h}_n]$ as output. Now, consider a function $msk(I, mp)$ that randomly masks $mp$ percent of tokens in $I$. The goal of the network is to predict the probability that a token $t_i$ corresponds to the target word $\hat{t}_i$ using the softmax function:

$$p(t_i = \hat{t}_i|I) = \frac{e^{\vec{w}_{\hat{t}_i} \cdot \vec{h}_i}}{\sum_{k=1}^{d} e^{\vec{w}_{t_k} \cdot \vec{h}_i}}$$

(1)

where $\vec{w}_{\hat{t}_i}$ is the weight vector of the linear layer for word $\hat{t}_i$. The loss of the ELM task is the cross entropy loss:

$$L_{ELM} = -\sum_{I \in D} \sum_{t_i \in msk(I, mp)} \log p(t_i = \hat{t}_i|I)$$

(2)

The second task is the Strand-Symbolic Mapping (SSM), in which the goal is to predict whether the symbolic expression is from the set of expressions representative of the strand (see Section III-C). To solve this task, we create both negative pairs by associating a strand with a random symbolic expression and...
positive pairs in which the symbolic expression is taken from its representative set. The ratio between positive and negative pairs is 50:50. An example of a positive pair can be seen in Fig. 2: the input strand computes the value 3 for register rbx, as stated by the corresponding symbolic expression. We believe that, with this task, the network is forced to learn the matching assembly snippets and symbolic expressions. This task is built by using a linear layer on top of the hidden state of the [CLS] token in the last layer that will be used to classify a pair as negative or positive. In mathematical terms, the goal of this task is to evaluate the probability that the output label is one, i.e., the symbolic expression correctly computes a value in the strand:

$$p(y = 1|I) = \frac{e^{w_0 h_{[CLS]}}}{e^{w_0 h_{[CLS]}} + e^{w_1 h_{[CLS]}}} \quad (3)$$

where $w_0$ and $w_1$ are the weight vector of the linear layer for label 0 (negative pair) and 1 (positive pair) respectively. The loss $L_{SSM}$ of the SSM task is the standard cross entropy loss.

The final loss on which BinBert is trained is the sum of the losses of the two tasks described above:

$$L = L_{ELM} + L_{SSM} \quad (4)$$

IV. Evaluation Tasks

The proposal of a new assembly model necessitates extensive experimental evaluation. In the field of NLP, standard multi-task benchmarks [29] are used to evaluate language models. However, equivalent benchmarks for binary code do not yet exist. Therefore, we designed our own benchmark by selecting several tasks at the levels of strands, CFG blocks, and functions. This approach tests BinBert on sequences beyond mere strands.

A. Intrinsic Tasks

The intrinsic tasks [30] directly use the embeddings produced by BinBert; there is no fine-tuning, and the embeddings are not used as input for other models.

1) Opcode Outliers: In the opcode outlier task we are given a set of five instructions. Four of these instructions belong to the same semantic class, and one is an outlier. For example, if the set is add eax, ebx; sub ebx, ecx; imul ecx, edx; add eax, 5; call printf, the last one is an outlier. The network has to predict which is the outlier among the five instructions.

2) Strand Similarity: In the strand similarity task we compute the embeddings of given strands with BinBert and use them to discover semantically similar strands. Two strands are similar if they have an overlapping semantic (non-empty intersection of the representative sets). More formally, we have a lookup database of $n$ strands $A = a_1, \ldots, a_n$ and a query strand $q$. The lookup contains strands that are similar to $q$ and strands that are dissimilar. Given a number $k$, the network has to return the $k$ strands in $A$ that are most similar to $q$.

B. Extrinsic Tasks

In the extrinsic tasks, BinBert will be used as the encoding layer of a neural architecture and fine-tuned end-to-end.

1) Strand Similarity: The extrinsic strand similarity is the same task as its intrinsic version; in this case, we fine-tune BinBert using a dataset of similar and dissimilar pairs of strands. We decided to include this task as it will quantify the effect of fine-tuning on the creation of semantic preserving embeddings.

2) Strand and Block Compiler Provenance: In the strand compiler provenance task, the architecture has to recognize the compiler and the optimization levels used to generate a particular strand. This task has been previously proposed on functions [5] and fragments of code [31]. In compiler provenance, the network has to recognize the syntactic signature that a compiler, or optimization level, produces. The block compiler provenance task is analogous but at the block level.

3) Strand Recovery and Execution: We designed two novel tasks that test the semantic understanding of assembly sequences. In strand recovery, we provide the network a basic block of the CFG where one instruction is marked. The DNN has to recognize all instructions in the same strand of the marked instruction. This task tests the understanding of the inputs/outputs of instructions, as the network has to infer the dependency created by implicit registers such as RFLAGS.

In strand execution, a strand and a question are given to the network. The question is composed of an assignment for the inputs and a marked output. The network has to predict the value of the marked output. This task is interesting as it forces the network to concretely execute the snippet of assembly code and compute the correct output.

4) Function Level - Compiler Provenance and Similarity: Finally, our multitask benchmark includes two tasks at the function level. In the function compiler provenance task, the network is tasked with analyzing an entire binary function to predict the compiler used to generate the function, as well as the optimization level. In the function similarity task, given a database of functions and a set of query functions, the network must identify, for each query, the similar functions in the dataset. We use the standard definition of function similarity [1]: two assembly functions are considered similar if they are derived from the same source code but compiled with different compilers or optimization levels. Function similarity is currently a prominent research topic [1], [2], [3], [14], [24] due to its significant security implications, which are discussed in depth in Section IX.

V. DATASETS, PRE-TRAINING AND IMPLEMENTATION DETAILS

Our datasets and the implementation of our system are released and open-sourced. They can be found at https://github.com/gadiluna/BinBert.

A. Datasets

We used three datasets, a pre-train dataset PTData used to pre-train BinBert, a test dataset TestData used for the downstream tasks, and another test dataset SimTestData specifically used for the similarity tasks. Our datasets are for X64.

1) PTData - Pre-Training Dataset: The pre-training dataset, PTData, includes the following projects: ccv-0.7, binutils-2.30, valgrind-3.13.0, libhttpd-2.0, openssl1.1.1-pre8, openmpi-3.1.1, coreutils-8.29, gsl-2.5, gdb-8.2, postgresql-10.4,
ffmpeg-4.0.2, and curl-7.61.0. These projects were compiled using compilers clang-3.8, clang-3.9, clang-4.0, clang-5.0, gcc-3.4, gcc-4.7, gcc-4.8, gcc-4.9, and gcc-5.0. For each compiler, we compiled every project four times, once for each optimization level in O0, O1, O2, O3.

We used radare2 (version 5.6.0) to extract function signatures and angr (version 9.1.11611) to obtain CFGs, basic blocks, and strands, as well as to derive the set of symbolic expressions from each strand. After removing duplicates, we obtained 17,215,046 pairs in the form \((\text{strand, simexpr})\) as dataset for the SSM task.

2) TestData - Test Dataset: The test dataset, TestData, is derived from projects diffutils-3.7, findutils-4.7.0, inetutils-2.0, mailutils-3.10, and wget-1.20.3. We used compilers clang-3.8, clang-6.0, clang-9, gcc-5, gcc-7, gcc-9, and icc-21, along with the four optimization levels O0, O1, O2, O3, to generate the raw binaries. These raw binaries will be used to create specific datasets for each task. As some operations are task-dependent, we discuss the specific split and format of the data in each task’s experimental section. We have ensured the removal of duplicates so that the same sample will not be present in both fine-tune and test splits.

3) SimTestData - Similarity Test Dataset: The similarity test dataset, SimTestData, comprises the following projects: putty-0.74, ImageMagick-7.0.10-62, sqliet-3.34.0, gmp-6.2.0, zlib-1.2.11, nmap-7.80 and libtomcrypt-1.18.2. Similar to the previous dataset, we used compilers clang-3.8, clang-6.0, clang-9, gcc-5, gcc-7, gcc-9, and icc-21, with the four optimization levels O0, O1, O2, O3. This dataset has been created to construct a large benchmark for testing various state-of-the-art solutions for similarity tasks.

4) Model Parameters, Pre-Training and Implementation Details: Our model is built using python 3, with pytorch (version 1.10.2+cu113) [32] and hugginface (version 4.16.02) [33]. We trained it on a DGX A100, using 4 A100 GPUs.

BinBert parameters are: sequence length 512, hidden size 768, intermediate size 3072, 12 attention heads and layers. The total number of parameters is 92,645,512. We used AdamOptimizer and learning rate 0.0001. The masking rate \(mp\) is 0.3. The batch size for each device is 32 with two steps of gradient accumulation; having 4 GPUs the equivalent batch size is 256. We trained for 1 epoch using 1425 steps of warmup. The time needed for the pre-training is 46 GPU hours.

VI. EXPERIMENTAL EVALUATION

In our evaluation we answer the following experimental questions:

- **RQ 1** Is an execution-aware transformer model trained on strands of assembly instructions and symbolic expressions the state-of-the-art assembly model for binary understanding?
- **RQ 2** What is the impact of pre-train on the performance across several binary understanding downstream tasks?
- **RQ 3** What is the impact of using a symbolic execution-aware pre-training? How it fares against the standard Masked Language Modeling (MLM) of Bert?

To answer RQ 1 we compare BinBert with PalmTree and Trex on all the tasks of our benchmark. We also compare BinBert with state-of-the-art function similarity solutions SAFE [1], GNN-BoW, GMN-BoW [34], [35] and BinShot [36] on the intrinsic function similarity task. For a fair comparison, we retrain both PalmTree, GNN-BoW and GMN-BoW on our pre-train dataset, by using the same parameters of the original papers. On all the extrinsic tasks we fine-tune PalmTree and Trex on exactly the same dataset we used to fine-tune BinBert. It is worth noticing that PalmTree authors highlighted the possibility of fine-tuning their system but they did not explore this possibility in their paper; while Trex authors test the fine-tuning on the function similarity task only. We are interested in assessing the effective contribution of the symbolic expressions used during pre-training (RQ 3) and the pre-training itself on downstream applications (RQ 2). To do so we will use the following baselines:

- **BinBert-MLM:** This is a model pre-trained on strands only with the standard Masked Language Modeling Task (MLM). In MLM only the assembly of a strand is given to the transformer during pre-training, the pre-training task is to recover masked tokens. The gap between this model and BinBert quantifies the impact of symbolic execution-awareness.
- **BinBert-FS:** This is a transformer encoder with the application-specific neural architecture trained from scratch on the specific downstream task. The gap between this model and BinBert quantifies the impact of pre-training.

For the fine-tuned models we use the notation: BinBert-FT, BinBert-MLM-FT, PalmTree-FT, and Trex-FT.

For each task, we will provide the details of the dataset, the solution we employed to solve the problem, the metrics used to evaluate the performance, and the final results.

A. Intrinsic Tasks

The results for intrinsic tasks are reported in Table II.

1) Opcode Outlier Detection:

   a) Dataset: We used 43618 different instructions to create a dataset of 50000 sets of 5 instructions each. These sets are created as we have defined in Section IV-A1. Analogously to [13], opcodes are categorized according to the \(x86\) Assembly Language Reference Manual. In Table III we reported the classes that we have used to categorise instructions for the opcode outlier detection. In particular, for each of the ten classes in Table III we created 5000 groups, in each group the outlier is chosen at random among all the instructions of the other classes. This ensures that we have a balanced dataset: for each possible pair of classes on average, we have the same number of groups in our dataset.

   b) Metrics and Results: To solve this task, we embed each instruction in the set and we evaluate if the embeddings are able to distinguish the outlier; this is done by computing the distance of each embedded instruction from the others and by predicting as outlier the most distant. An instruction embedding is computed by mean pooling the instruction tokens’ hidden states in the second last layer of BinBert (we take the second

---

[3]https://docs.oracle.com/cd/E26502_01/html/E28388/ennbz.html

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
last layer as it is less influenced by the pre-training task). The evaluation metric is the accuracy of [30]:

\[
    \text{Accuracy} = \frac{\sum_{s \in S} \text{outlier}(s)}{|S|}
\]

where \(S\) is the dataset composed of instruction sets and \(\text{outlier}(s)\) is equal to 1 if the outlier in the instruction set \(s\) is detected and 0 otherwise.

We compute the mean accuracy and standard deviation on 10 runs of the experiment on different datasets (each composed of 50 k sets), the results are in Table II. BinBert achieves the best performances (81.9 accuracy) and it shows a great improvement over BinBert-MLM (71.9 accuracy): this confirms that symbolic expressions clearly enrich the semantic learned by the model for each instruction. It also outperforms PalmTree and Trex (77.6 and 75.7 accuracies respectively) by a wide margin. The performances of PalmTree are not dominated by a transformer trained on MLM. We believe that this is so because PalmTree has been explicitly designed to be an instruction embedding solution, while BinBert-MLM has been trained on sequences. Moreover, Trex outperforms BinBert-MLM, meaning that micro-execution traces used by Trex enhance the semantic of each instructions. However, Trex does not outperform BinBert: thus confirming that symbolic expressions are more powerful than concrete execution values used by Trex. In Section VIII-A we report a qualitative analysis with the clusters of opcodes learned by BinBert.

2) Similarity at Strand Level:

a) Dataset: We use a database \(A\) of 42547 strands on which we perform 4942 queries \(Q\) which correspond to the number of equivalence groups. That is we perform one query for each group of similar functions. On average for each query, we have 8.61 (s.d. ±4.9) similar elements in \(A\).

b) Metrics and Results: To solve the task we compute an embedding vector \(\vec{q}\) for each query \(q\) and an embedding vector \(\vec{a}\) for each strand \(a\) in the lookup database \(A\). This is done by averaging all the instruction tokens’ hidden states in the second last layer of BinBert. For each query vector \(\vec{q}\) we compute the cosine similarity with all \(\vec{a} \in A\); we return the ordered list of the top-k similar elements \(R_q = (r_1, \ldots, r_k)\). Using \(R_q\) we compute: precision, number of true similars in \(R_q\) over \(k\); recall, number of true similar in \(R_q\) over \(\#\text{sim}(q)\), that is the number of items similar to \(q\) in \(A\); and nDCG. The nDCG is a measure used in information retrieval. It is defined as:

\[
    nDCG = \frac{\sum_{i=1}^{k} \frac{\text{sim}(r_i, q)}{\log(1+i)}}{\sum_{i=1}^{\#\text{sim}(q)} \frac{1}{\log(1+i)}}
\]

where \(\text{sim}(r_i, q)\) is 1 if \(q\) is similar to \(r_i\) and 0 otherwise. The quantity at the denominator is the scoring of a perfect answer, and the number at the numerator is the scoring of our system. The nDCG is between 0 and 1, and it takes into account the ordering of the items in \(R_q\), giving better scores when similar items are ordered first. As an example let us suppose we have two results for the same query: (1,1,0,0) and (0,0,1,1) (where 1 means that the corresponding index in the result list is occupied by a similar item and 0 otherwise). These results have the same precision (i.e., \(\frac{1}{2}\)), but nDCG scores the first better. We average the per-query precision, recall, and nDCG to obtain the final metrics. Results are shown in Table II and in Fig. 3. We can see that BinBert and BinBert-MLM achieve the best performance on precision, recall and nDCG, the MLM model has a slight edge on BinBert.
B. Extrinsic Tasks at Strand and Basic Block Level

Results for extrinsic tasks are shown in Table IV.

1) Strands Similarity:

a) Dataset: We fine-tuned BinBert on the task of recognizing whether strands are similar or dissimilar. We created a dataset of 49974 samples, each consisting of an triplet consisting of an anchor strand \(a\), a positive strand \(p\) (similar to the anchor), and a negative strand \(n\) (dissimilar from the anchor). We split it into train and validation, resulting in 39978 strands triplets for the training set and 9996 for the validation. We test the fine-tuned models on the same test sets used in the corresponding intrinsic tasks (see Section VI-A).

b) Fine-tuning: We fine-tune the model using a siamese architecture [1], [37], [38] with the triplet objective function [39] (see Fig. 4). In this architecture three instances of the embedding network are used, each instance produces the embedding of the corresponding entity in a triplet. The resulting embeddings are used during training to minimize the triplet margin loss that has the following mathematical form:

\[
\mathcal{L}(\vec{a}, \vec{p}, \vec{n}) = \max\{||\vec{a} - \vec{p}||_2 - ||\vec{a} - \vec{n}||_2 + \epsilon, 0\}
\] (7)

The training process instructs the embedding network to produce embeddings such that the euclidean distance between \(\vec{a}\) and \(\vec{p}\) is smaller than the euclidean distance between \(\vec{a}\) and \(\vec{n}\) by at least a margin of \(\epsilon\). For BinBert-FT and BinBert-MLM-FT the embeddings for the triplet loss are given by the average of all tokens of the last layer (excluded the padding).

Differently from BinBert, which can directly embed any arbitrary sequence of assembly code, Palmtree is an instruction embedding model, thus requiring an additional architecture to transform it into a model that embeds sequences. In particular, we use a bidirectional LSTM, where each cell takes as input the instruction embedding produced by PalmTree. Finally, we compute an embedding by averaging all the hidden states of the LSTM. We stress that in case of BinBert we do not need to use an LSTM to compute a function embedding but we use the average of the tokens of the last layer (excluded the padding).

Regarding Trex, we used the same finetuning process used in the original paper for the function similarity test.
We fine-tune each model for 20 epochs, selecting the epoch with the best AUC on validation.

c) Metrics and Results: Results for strands similarities are in Table IV. In Fig. 5 there are the results of strand similarity for \( k \in \{0, 50\} \).

Also in this case BinBert achieves the best performances. The gap between BinBert-FT and the other models is much wider than in the intrinsic case. A possibility is that BinBert learns a wider semantic during pre-training, solving more efficiently the similarity task after fine-tuning. This explains the gap between BinBert-FT and BinBert-MLM-FT.

The great impact of pre-training can be appreciated by looking at the performance of BinBert-FT and BinBert-FS. BinBert-FS has been only trained on the fine-tune dataset so it cannot leverage a learned semantic, the fine-tune dataset has not enough data to make up for this disadvantage and to train a big transformer model.

2) Compiler Provenance: As in other works on compiler provenance [5], we train and test the networks on the task of Compiler Classification, that is detecting the compiler family that has generated a sample, and Optimization Classification, detecting the optimization level used to generate a sample.

a) Dataset: The compiler provenance dataset is made up of samples in which, depending on the granularity, a strand or a basic block is associated with two labels: the compiler family with its corresponding version and the optimization used. The strand compiler provenance dataset has 30760 samples, split into 24564 samples for the training set, 3104 samples for the validation set, and 3092 samples for the test set. Each compiler family contains 4394.29 samples on average (s.d. ±185.08), thus resulting in a balanced dataset. In terms of optimization levels, there is an imbalance similar to the strand case. In fact, the O0 optimization level contains 26380 samples, while other levels contain 15624 samples (s.d. ±1828.63).

b) Fine-Tuning: We fine-tune BinBert, BinBert-MLM, and Trex on both compiler and optimization classification by adding a linear layer followed by the softmax function on top of the last layer hidden state corresponding to the [CLS] token. For PalmTree we use an LSTM over the instruction embedding tokens generated; to obtain a classifier we attach a linear layer with the softmax on top of the last hidden states of the RNN. We fine-tune each model for 20 epochs, selecting the epoch with the best classification accuracy on validation.

c) Metrics and Results: To assess the performance of the models, we utilized macro precision, recall, and F1 score. These metrics guarantee equal treatment of all classes, thereby mitigating the potential for misleading outcomes in the presence of imbalances as discussed earlier. Results for both compiler and optimization classification are shown in Table IV. In this case, BinBert-FT is slightly worse than BinBert-MLM-FT, we believe that this is due to the fact that recognising a compiler signature is a syntactic task. PalmTree-FT and Trex-FT have the worse performance among all fine-tuned models. Again we can see that pre-training is important as the from-scratch model performs consistently worse than its fine-tuned counterpart. Results are similar for the optimization classification task. Since the optimization dataset has some imbalances, we reported in Fig. 6 the confusion matrices obtained by using BinBert on the optimization classification tasks for both strands and basic blocks. We can observe that BinBert is better at distinguishing optimized (O1, O2, O3) vs unoptimized code (O0) rather than recognizing the specific optimization level used to compile it.

3) Strand Recovery:

a) Dataset: The dataset is made of 9265 basic blocks, each block contains at least 5 disjoint strands (i.e. the strands do not overlap on instructions). We split the dataset into 7411 training samples and 927 samples for both validation and test. We model strand recovery as instruction classification; given the instructions of a basic block and the final instruction of one of its strands, we aim at classifying the other instructions as either belonging to the same strand as the marked instruction
or not. Thus, the classification of each instruction is a binary classification task. To mark an instruction we surround it with a special token. The total number of instructions is 105288, 13230, and 13138 for the training, validation, and test set respectively. The total number of instructions belonging to the positive class is 29% in each set.

b) Fine-tuning: We fine-tune BinBert, BinBert-MLM, and Trex by attaching a classification head on top of the last layer hidden states of the first token of each instruction; the network will output 1 if an instruction is part of the strand to be recovered and 0 otherwise. As for previous tasks, we use an LSTM on top of PalmTree and we put a classification head on the hidden states of the first token of each instruction. We fine-tune each model for 20 epochs, selecting the epoch with the best classification accuracy on validation.

c) Metrics and Results: To evaluate the performance of the models, we used precision, recall, and F1-score of the positive class, which is the minority class as well. Results are shown in Table IV. We can see that the best-performing model is BinBert-FT, it is outperformed only by Trex-FT in terms of precision, but it markedly surpasses PalmTree-FT on all the metrics considered. BinBert-MLM-FT achieves the same precision as BinBert-FT but smaller recall. We believe the reason to be the execution-awareness of BinBert-FT that allows the model to recover more instructions in a strand. For the sake of completeness, we reported the performance of BinBert on the negative class as well: 99.6 Precision, 99.0 Recall and 99.3 F1 score. A qualitative analysis showing the change of internal attention weights of BinBert during fine-tuning is in Section VIII-B.

4) Strand Execution:

a) Dataset: We created a dataset for the strand execution task by taking strand-symbolic execution pairs, assigning concrete values to input variables, and evaluating its concrete output. In particular, we randomly assign values between 0 and 100 to input variables and we take only strands whose output is not greater than 200. Our dataset only contains strands computing the value of a register or a predicate of a conditional branch. The dataset is composed of 40000 training samples, and 5000 validation and test samples (total 50 k). In each dataset, approximately 83% of the samples have an output value below 100. For each output value between 0 and 100 there is an average number of 41 strands (s.d. ±7.65).

Each sample is made up of strand instructions followed by concrete assignments of input variables and the query output variables (only in the case of register outputs); the label for such a sample is the concrete value of the output variable. For instance, consider the strands `mov eax, dword ptr [rbp - 180] sub eax, 1` and the value 9 assigned to `dword ptr [rbp - 180]`. The corresponding sample will be `mov eax, dword ptr [rbp - 180] sub eax, 1 [SEP] dword ptr [rbp - 180] = 9 [SEP] rax`. The network has to predict the value for register rax, in this case 8.

b) Fine-tuning: We model this problem as a sequence classification task, thus we used the same architectures used for the compiler provenance tasks (see Section VI-B2). Regarding Trex, since it uses a bi-LSTM to deal with concrete values, we pass our concrete input inside that LSTM architecture.

c) Metrics and Results: To evaluate the performance of the models and to address dataset imbalances we used macro precision, recall and F1 measures. We reported results obtained by all the models in Table IV. BinBert-FT achieves the best performances. PalmTree-FT and Trex perform poorly on this task. In the case of PalmTree we believe that this is due to the fact that it is a pure instruction embedding model so even when fine-tuned it cannot transfer to the upward neural architecture A sequence-related knowledge. On the contrary, both BinBert-FT and BinBert-MLM-FT are trained on sequences; however, BinBert-FT has the edge thanks to its execution-awareness. Regarding Trex, we believe that low performances are due to its architecture that treats concrete values in a separate network. In this case, the pre-training has a great impact as we can see by the poor performance of BinBert-FS.

C. Extrinsic Tasks at Function Level

To address function-level tasks, we employ BinBert on the Linearized CFG. This approach serves two purposes. First, it evaluates whether the drawbacks of such a representation, as discussed in Section II-A2, are offset by the pre-training phase. During this phase, the noise from unrelated instructions is filtered out through the use of strands. Second, it aims to
assess the representational capacity of our assembly code model independently of complex architectures or elaborate function representation. This is crucial for determining that the source of performance improvements stem from the assembly code model itself and are not due to other structures, either integrated with or built upon our model, which could confound the evaluation of its true effectiveness.

1) Similarity:

a) Fine-Tune Process and Dataset: The function similarity dataset has exactly the same format as the similarity tasks at strand level (see Section VI-B1). In particular, the training dataset contains 18834 function triplets, while the validation set contains 7650 triplets.

On top of the embedding models, we use the same architectures that we employed to solve the other similarity tasks (see Section VI-B1). Specifically, for PalmTree we use it on the linearized CFG using an LSTM on top of the instruction embeddings.

We compare our solution with some other approaches that have been specifically designed to tackle the binary similarity problem. These include SAFE [1], Trex [14], GNN-BoW, GMN-BoW [34], [35] and BinShot [36]. SAFE uses word2vec [11] to create instruction embeddings that are then fed to a self-attentive neural network [40] for learning the final embeddings for functions. Binshot utilizes a transformer encoder that has been pretrained on the MLM task and is subsequently fine-tuned using pairs of similar and dissimilar functions. BinShot calculates a weighted distance vector from a given pair of functions and then employs a fully-connected layer on this vector to output a final similarity score. GNN-BoW and GMN-BoW are both based on graph neural networks and are the best-performing solutions for the binary similarity problem according to [35].

In these approaches, the function control flow graph (CFG) is represented as a graph, and the nodes are represented by the bag of words (BoW) of the 200 most frequent opcodes contained in each block. While GNN-BoW computes the embedding of each function independently and then computes the similarity score, GMN-BoW computes the embeddings of a pair of graphs simultaneously. However, even though GMN-BoW achieves the best results, it cannot be used in some scenarios due to its time performance. This is because, unlike traditional embedding models, the function embeddings cannot be precomputed since they depend on the query function. This makes large-scale testing infeasible. Similar observations can be done for BinShot, in which the model is designed to produce a similarity score between function pairs rather than outputting function embeddings. For this reason, we performed two experiments on different datasets. The first experiment was conducted on a larger dataset, where all models were evaluated except GMN-BoW and BinShot. This dataset comprised 58773 functions, with 5000 of them serving as queries (the number of queries corresponds to the number of equivalence groups). On average, each query had 11.7 similar entities (s.d. ±5.29). The second experiment was conducted on a reduced dataset, where GMN-BoW and BinShot took a reasonable amount of time to compute. This dataset consisted of 5962 functions, with 422 of them serving as queries. On average, each query had 15 similar entities (s.d. ±5.91).

b) Results: Results are in Fig. 7 and Table IV. BinBert achieves the best performances, beating also GNN-BoW, SAFE and Trex which are specific for the function similarity problem. PalmTree performs worse than BinBert, confirming our initial intuition about the drawbacks of the lack of context and the isolated instruction embeddings. SAFE has lower performances than PalmTree. This suggests that even if they are both based on fixed instructions embeddings, a transformer encoder is more capable of capturing an instruction semantic than word2vec. Unsurprisingly, the Trex model beats PalmTree. Also in this task, BinBert-FS is below its corresponding fine-tuned version highlighting the importance of pre-training. The impact of execution awareness can be seen in the advantage of BinBert-FT on BinBert-MLM-FT. Finally, the results for the experiments on the reduced dataset are in Fig. 8, also in this case we can see that BinBert-FT is the state-of-the-art and it performs better than GMN-BoW and BinShot.

1) Compiler Provenance:

a) Fine-tune Process and Dataset: The compiler provenance dataset has the same structure as the strand and block compiler provenance tasks. It contains 39017 functions in the training set and 4877 functions in the validation and test set. Each compiler family contains 6967.42 samples on average, thus resulting in a balanced dataset (s.d. ±1600.42). Regarding optimization levels, there exists an imbalance similar to the strand and basic block cases. In fact, the O0 optimization level contains 16258 samples, while other levels contain 10838 samples (s.d. ±2589.33).

We used the same architectures that we employed to solve the other compiler provenance tasks.

b) Results: We reported the macro precision, recall, and F1 score for different models in Table IV. Results are similar to the compiler provenance task at strand and block level, thus the same considerations hold. In Fig. 9 we reported the confusion matrices obtained by using BinBert on the compiler and optimization classification tasks. We can observe that BinBert can clearly distinguish among different compiler families and it only gets confused with different versions in the same family. Similar behavior can be observed in the optimization classification task. It is easier for BinBert to distinguish optimized (O1, O2, O3) vs unoptimized code (O0) than to recognize the specific optimization level used to compile it.

Results from the experimental evaluation confirm that BinBert is the current state-of-the-art for assembly code models. It shows improvement over PalmTree, Trex, and other deep learning solutions specifically tailored for a certain task. Execution awareness has a marked impact on semantic tasks. Interestingly, on some syntactic tasks, execution awareness does not increase the performance of the model. Our evaluation highlights the great impact of pre-training on downstream tasks with small-size datasets.
In all of the aforementioned experiments, we utilized the Wordpiece tokenizer, which is a standard choice for Bert-like models. However, we also assessed the performance of the model using two additional tokenization strategies: whitespace tokenizer and unigram tokenizer. The whitespace tokenizer, employed by PalmTree and Trex, involves splitting tokens based on whitespaces and symbols. The unigram tokenizer initializes its vocabulary with a large number of symbols and progressively trims it down to obtain a smaller vocabulary. Our experimental evaluation shows that the whitespace tokenizer results in an average performance decrease of 1.88%, while the unigram tokenizer leads to a performance decrease of 8.91%.
TABLE V
RESULUTS FOR TIME ANALYSIS

| Model       | Time (s) |
|-------------|----------|
| BinBert     | 37.3     |
| BinBert-MLM | 37.3     |
| Palmtree    | 49.7     |
| Trex        | 87.9     |
| SAFE        | 37.6     |
| GNN-BoW     | 19.1     |
| GMN-BoW     | 117.65.8 |
| BinShot     | 2327.9   |

VII. TIME PERFORMANCE COMPARISON

Table V shows the time performance of different solutions for binary similarity. The measures, in seconds, are the time required for the models to generate embeddings and find the top k most similar functions in the reduced dataset. The GNN-BoW model is the fastest, likely due to its smaller size of basic block embedding. Palmtree is slower than BinBert and SAFE, possibly due to the need of applying a transformer model separately to each assembly instruction. Trex is slower than Palmtree due to its use of an additional LSTM architecture to embed numerical values. GMN-BoW is the slowest model, as it must generate a similarity score for each pair of functions in the dataset. The performance of GMN-BoW may not be practical in a real-world scenario, especially when the network has to be used to retrieve similar functions in a large database of samples. BinShot outperforms GMN-BoW in terms of speed, but it lags behind other embedding models due to the necessity of computing similarity scores for each function pair.

VIII. QUALITATIVE ANALYSIS OF BINBERT

We performed a qualitative analysis of the BinBert model by examining the clusters of opcodes created by the model and by visualising its attention mechanism.

A. Opcode Clustering

We performed a qualitative analysis based on the visualization of the clusters of opcodes learned by BinBert. To do so, we first used BinBert to convert each opcode into a vector, and we then applied t-SNE [41] to visualize opcodes in a two-dimensional space. Results are shown in Fig. 10. Opcodes are well clustered according to their semantics. Specifically, we can identify two symmetric regions in which jumps, conditional move, and conditional set are split based on the condition checked (either positive or negative). We can also identify a region containing operations performing multiplications (imul, lea) and divisions (divq), and other arithmetic operations (add, sub). Another interesting example is given by the region containing ret, pop, push and call: they all manipulate the stack.

B. BinBert Attention Visualization

BinBert is based on a stacked attention mechanism, in each layer of the transformer, there is an attention mechanism that creates the embedding of a token by focusing on the most interesting part of the input layer. We used BertViz [42] to visualize the attention weights produced by BinBert at certain layers and heads. Specifically, we were interested in evaluating two aspects: the robustness of the attention when BinBert is fed with assembly sequences that are not strands and the effect of finetuning on those weights. We fed BinBert with a strand (Fig. 11(a)). We observe the last attention head in the last layer, and we discover that when computing the embedding of the call instruction, BinBert focuses on the parameters of the call which are 3 and 4 (Fig. 11(a)) and on the name of the called, meaning that it has understood the strand relationship and it understood the calling convention. We then added an instruction (inc rbx) to the previous strand and we gave it as input to BinBert. We can see, that the attention is still focusing on the function parameters, but it is also considering the rbx register of the outlier instruction. We then feed the same non-strand instruction to BinBert-FT finetuned on the strand recovery task. We observe that finetuning has lowered the attention weight associated with the rbx register, thus reducing the noise. This last observation shows that during fine-tuning BinBert is able to reconfigure its
Fig. 11. Series of Attention Visualization snapshots, we can see that the network changes its attention paradigm during fine-tuning to more effectively tune down the noise inserted by extraneous instructions.

attention mechanism according to the downstream task to be solved.

IX. SECURITY APPLICATIONS OF AN ASSEMBLY CODE MODEL

The BinBert assembly code model is suitable for various security applications. The model transforms sequences of assembly instructions into representations that other machine learning models or neural architectures can directly use and can be seamlessly integrated into existing solutions to leverage the semantic knowledge acquired during pre-training. In this section, we show possible venues of applications of our model.

A. Reverse Engineering

An assembly code model can assist and automate multiple tasks beneficial for reverse engineers. In this scenario, a fundamental challenge is understanding the semantics of unknown functions. A valuable tool is a high-level representation in the form of decompiled code. To produce decompiled code from a low-level programming language, some deep learning-based solutions use encoder-decoder architectures [43], where BinBert could replace the encoder part. Additionally, a code model can automate the task of naming assembly functions with semantically representative names. Recent studies [44], [45], [46] suggest neural architectures for this challenge, where many process sequences of assembly instructions using Transformers or RNNs as encoders. BinBert can be directly integrated into such solutions either as an encoder (for example, in [44], [46]) or as a method for creating feature vectors from assembly instructions (for example, in [10], [47]). Another area where an assembly code model proves useful is binary authorship detection, aiming to identify the author of an executable and, in malware cases, the Advanced Persistent Threat (APT) group responsible for the attack [48].

The compilation and optimization classification tasks are also significant from a security perspective. It has been recognized that specific versions of compilers and optimization levels may inadvertently introduce security vulnerabilities into cryptographic code [49]. An assembly model can furnish analysts with insights into the probable optimization level utilized for compiling a binary. This information could steer their analysis towards identifying particular security-sensitive bugs.

B. Binary Similarity

The Binary Similarity problem is a fundamental problem that has application in a wide range of security-related tasks [35]. As an example, it has been used to classify unknown malware in families [1], or to find known vulnerabilities (CVE) in specific binaries used in production environments [14], [36], [38]. This aspect is especially important in contexts with vulnerable firmware, where a flaw at the source code level can affect numerous devices across different architectures.

X. RELATED WORKS

Binary code representation techniques can be subdivided into two main branches: manual features selection [10], [38], [47], [50], [51], [52] and unsupervised features extraction. Since our paper proposes an assembly model, we discuss works that use or propose instruction embedding models. We first focus on the instruction embedding models proposed in the literature, categorising them according to the defining properties identified in the Background Section II: the distributed representation learning used, the preprocessing of assembly instructions, and the extraction methodology for assembly sequences. Finally, we discuss how these models have been used to solve binary analysis tasks.
A. Distributed Representation Learning

The distributed representation learning technique is the neural architecture and the training tasks used by the instruction models to create useful embedding vectors. The most common is word2vec [11]. Word2vec has been used, with minimal modifications, by Eklavya [12], SAFE [1] and others [2], [5], [24]. A notable difference is Asm2vec [25] which uses PV-DM [21], a variation of word2vec that simultaneously creates instructions and function embeddings. We remark that PV-DM cannot be fine-tuned. PalmTree [13], Trex, Stateformer and Binshot, that we described in Section II-A3, use a transformer architecture, such architecture contain an embedding layer that automatically learn a distributed representation of the input tokens.

B. Preprocessing of Assembly Instructions

The preprocessing is characterised by the substitution policy for information in the raw assembly instruction and the tokenization policy. We classify the substitution policies in aggressive or light. In an aggressive policy, lots of information contained in the assembly instructions are removed or changed. An example is DeepBinDiff [24] that replaces all constants and pointers with special tokens and renames registers according to their lengths in bytes (e.g. ecx becomes reg4). InnerEye [2], instead, replaces all constants, strings, and function names with special symbols. BinDeep [53] substitutes operands based on predefined categories (e.g. general register, direct memory reference, etc.). All the above policies are aggressive; indeed, InnerEye [2] wastes fundamental information that a library function call could bring and DeepBinDiff [24] loses register names that could be relevant, for instance, to understand the data flow.

Light preprocessing policies are applied by SAFE [1] and PalmTree [13] which keep small constant values and replace values above a predefined threshold with a special token. SAFE [1] replaces all memory addresses with the same token, PalmTree uses different tokens for generic memory locations and the ones pointing to strings.

Regarding tokenization, some works consider an entire instruction as a token [1] or split an instruction into opcode and operands [25], while others use a word-based tokenizer by splitting instructions on symbols (e.g. spaces or special characters) [14], [13]. The latter is a fine-grained tokenization strategy that is useful to reduce the vocabulary size and allow the upstream network to separately learn the semantics of each token (mnemonics, registers, etc.). No one used automated tokenization strategies like WordPiece [26].

C. Binary Analysis Solutions Using Embedding Models

Deep-learning-based solutions can be categorized into approaches using Graph Neural Networks (GNN), Recurrent Neural Networks (RNN), and Transformer architecture. [5], [9] use a GNN applied to function control flow graphs after transforming each block into a vector representation; this transformation is done by aggregating the instruction embeddings of each block. The resulting architecture is applied to the binary similarity [5], the compiler provenance problem [5], function naming problem [9]. Eklavya [12] and InnerEye [2] are examples of RNN-based solutions applied to the recovery of arguments used by a binary function [12], and basic-block similarity [2]. Another work is SAFE [1] which added a self-attention layer on top of an LSTM to solve the binary similarity problem. Among transformer-based solutions, we can find [54], [9] that apply transformers encoder-decoder architecture to recover function names from stripped binaries. Other works [3], [14], [36] use transformer architecture and specifically tackle the binary similarity problem. In particular, [3] uses a transformer encoder to obtain basic blocks embeddings to be fed into a GNN which is trained in conjunction with a Convolutional Neural Network (CNN) to produce a function embedding for binary similarity. Unfortunately, [3] does not release the code and the paper lacks of the details needed to reimplement their proposal.

XI. Conclusion

We presented BinBert, an execution-aware assembly language model. BinBert is trained on a big dataset of assembly strands and symbolic expressions. BinBert has shown state-of-the-art performance highlighting the relevance of execution-awareness. Our evaluation shows that BinBert is an encoder model that can be used to solve several tasks related to binary analysis using a fine-tune dataset of relatively small size. The generality of the model is an important strength and we believe that BinBert can be fruitfully applied to other downstream tasks as encoder layer of complex neural networks.

REFERENCES

[1] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, “Function representations for binary similarity,” IEEE Trans. Dependable Secure Comput., vol. 19, no. 4, pp. 2259–2273, Jul./Aug., 2022.
[2] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, “Neural machine translation inspired binary code similarity comparison beyond function pairs,” in Proc. 26th Netw. Distrib. Syst. Secur. Symp., 2019, Art. no. 50.
[3] 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 Proc. 24th AAAI Conf. Artif. Intell., 2020, pp. 1145–1152.
[4] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao, “HMALIA: Recovering compiler optimization levels from binaries by deep learning,” in Proc. Intell. Syst. Appl., K. Ari, S. Kapoor, and R. Bhatia, Eds, 2018, pp. 35–47.
[5] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, “Investigating graph embedding neural networks with unsupervised features extraction for binary analysis,” in Proc. Workshop Binary Anal. Res., 2019, pp. 1–11.
[6] D. Pizzolotto and K. Inoue, “Identifying compiler and optimization options from binary code using deep learning approaches,” in Proc. IEEE Int. Conf. Softw. Maintenance Evol., 2020, pp. 232–242.
[7] E. C. R. Shin, D. Song, and R. Moazzez, “Recognizing functions in binaries with neural networks,” in Proc. 24th USENIX Conf. Secure. Symp., 2015, pp. 611–626.
[8] C. Fu et al., “Coda: An end-to-end neural program decompiler,” in Proc. 53rd Int. Conf. Neural Inf. Process. Syst., 2019, pp. 3703–3714.
[9] Y. David, U. Alon, and E. Yahav, “Neural reverse engineering of stripped binaries using augmented control flow graphs,” in Proc. ACM Program. Lang., vol. 4, pp. 1–28, 2020.
[10] J. Patrick-Evans, M. Danniell, and J. Kinder, “XFL: Extreme function labeling,” 2021. [Online]. Available: https://arxiv.org/abs/2107.13404
[11] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in Proc. 26th Int. Conf. Neural Inf. Process. Syst., 2013, pp. 3111–3119.
A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A short code fragment," in Proc. 33rd Int. Conf. Neural Inf. Process. Syst., 2021, pp. 690–702.

A. Vaswani et al., "Attention is all you need," in Proc. 31st Int. Conf. Neural Inf. Process. Syst., 2017, pp. 6000–6010.

Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation, 2016, pp. 266–280.

Y. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in Proc. Conf. Empirical Methods Natural Lang. Process., 2014, pp. 1532–1543.

P. Bojanowski, É. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," 2016, arXiv:1607.04606.

Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in Proc. 31st Int. Conf. Int. Mach. Learn., 2014, pp. II-1188–II-1196.

W. Chen, Y. Su, Y. Shen, Z. Chen, X. Yan, and W. Y. Wang, "How large a vocabulary does text classification need? A variational approach to vocabulary selection," in Proc. Conf. North Am. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol., 2019, pp. 3487–3497.

Z. Hu, T. Chen, K.-W. Chang, and Y. Sun, “Few-shot representation learning for out-of-vocabulary words,” in Proc. 57th Annu. Meet. Assoc. Comput. Linguistics, 2019, pp. 4102–4112.

Y. Duan, X. Li, Z. Wang, and H. Yin, "DeepBinDif: Learning program-wide code representations for binary differencing," in Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp., 2020, pp. 23–26.

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 Proc. IEEE Symp. Secur. Privacy, 2019, pp. 472–489.

Y. Wu et al., “Google’s neural machine translation system: Bridging the gap between human and machine translation,” 2016, [Online]. Available: http://arxiv.org/abs/1609.08144

Y. Shoshitaishvili et al., "SOK: (state of) the art of war: Offensive technologies for ground combat," in Proc. 31st USENIX Secur. Symp., 2015, pp. 138–157.

A. Conneau and G. Lample, "Cross-lingual language model pretraining," in Proc. 33rd Int. Conf. Neural Inf. Process. Syst., 2019, pp. 7057–7067.

A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, “GLUE: A multi-task benchmark and analysis platform for natural language understanding,” in Proc. EMNLP Workshop BlackboxNLP Analyzing Interpreting Neural NLP, 2018, pp. 353–355.

J. Camacho-Collados and B.Navigli, "Find the word that does not belong: A framework for an intrinsic evaluation of word vector representations," in Proc. 1st Workshop Evaluating Vector-Space Representations NLP, 2016, pp. 43–50.

Y. Otsuka, O. Tsukasa, M. Mimura, T. Sakaki, and H. Uekiwa, "O-glassesX: Compiler provenance recovery with attention mechanism from a short code fragment," in Proc. Workshop Binary Anal. Res., 2020.

A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in Proc. Adv. Neural Inf. Process. Syst., 2019, pp. 8024–8035.

T. Wolf et al., "Transformers: State-of-the-art natural language processing," in Proc. Conf. Empirical Methods Natural Lang. Process. Symp. Demonstrations, 2020, pp. 38–45.

Y. Li, C. Gu, T. Dullien, O. Vecina, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in Proc. 36th Int. Conf. Mach. Learn., 2019, pp. 3835–3845.

A. Marcelli, M. Graziano, X. Ugarte-Pedrozo, Y. Fratantonio, M. Mancini, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in Proc. 31st USENIX Secur. Symp., 2022, pp. 2099–2116.

S. Ahn, S. Ahn, H. Koo, and Y. Paek, “Practical binary code similarity detection with BERT-based transferable similarity learning,” in Proc. 38th Annu. Comput. Secur. Appl. Conf., 2022, pp. 361–374.

J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, “Signature verification using a ‘siamese’ time delay neural network (nips’93),” in Proc. 9th Adv. Neural Inf. Process. Syst., 1996, pp. 737–744.

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. Commun. Secur., 2017, pp. 363–376.

N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese bert-networks," in Proc. Conf. Empirical Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process., 2019, pp. 3980–3990.

Z. Lin et al., “A structured self-attentive sentence embedding,” 2017. [Online]. Available: https://arxiv.org/abs/1703.03130

L. van der Maaten and G. Hinton, “Visualizing data using t-SNE,” J. Mach. Learn. Res., vol. 9, no. 86, pp. 2579–2605, 2008.

J. Vig, "A multiscale visualization of attention in the transformer model," in Proc. 57th Annu. Meet. Assoc. Comput. Linguistics Syst. Demonstrations, 2019, pp. 37–42.

Y. Cao, R. Liang, K. Chen, and P. Hu, "Boosting neural networks to decompile optimized binaries," in Proc. 58th Annu. Comput. Secur. Appl. Conf., 2022, pp. 508–518.

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. Commun. Secur., 2022, pp. 1631–1645.

H. Gao, S. Cheng, Y. Xue, and W. Zhang, "A lightweight framework for function name reassignment based on large-scale stripped binaries," in Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Anal., 2021, pp. 607–619.

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. Commun. Secur., 2023, pp. 951–965.

J. Patrick-Evans, L. Cavallaro, and I. Kindler, "Probabilistic naming of functions in stripped binaries," in Proc. 36th Annu. Comput. Secur. Conf., 2020, pp. 373–385.

S., Y. Zhang, L. Ouyang, and Y. Chen, “BinMLM: Binary authorship verification with flow-aware mixture-of-shared language model,” in Proc. IEEE Int. Conf. Softw. Anal. Evol. Reengineering, 2022, pp. 1023–1033.

V. D’Silva, M. Payer, and D. X. Song, "The correctness-security gap in compiler optimization," in Proc. IEEE Symp. Secur. Privacy Workshops, San Jose, CA, USA, May 21–22, 2015, pp. 73–87.

C. Liangboonprakong and O. Sornil, "Classification of malware families based on n-grams sequential pattern features," in Proc. IEEE 8th Conf. Inf. Electron. Appl., 2013, pp. 777–782.

D. Arp, M. Sprenztenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp., 2014, pp. 23–26.

B. M. Padmanabhan and H. B. K. Tan, "Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning," in Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf., 2015, pp. 450–459.

S. Afrabce, K.-K. R. Choi, M. Qusay, and M. Khasawneh, "BinDeep: Binary to source code matching using deep learning," in Proc. IEEE 20th Int. Conf. Trust Secur. Privacy Comput. Commun., 2021, pp. 1100–1107.

F. Artuso, G. A. Di Luna, L. Massarelli, and L. Quezaroni, “Function naming in stripped binaries using neural networks,” 2019. [Online]. Available: http://arxiv.org/abs/1912.07946

Fiorella Artuso received the master’s degree in computer science engineering, in 2019. She is currently working toward the PhD degree in computer science engineering with the Sapienza University of Rome. Her research interests include the application of deep learning and natural language processing techniques to source or binary code to solve problems in the software engineering and cyber security fields.
Marco Mormando received the master’s degree in computer science engineering, in 2021. He is a research assistant with Sapienza University of Rome. His research field is deep learning applied to cyber security. His recent projects consist of employing natural language processing techniques to address the binary similarity problem.

Giuseppe Antonio Di Luna received the PhD degree from the Sapienza University of Rome, in 2015, with a thesis on counting in anonymous dynamic networks. After the PhD degree, he did a postdoctoral research with the University of Ottawa, working on fault-tolerant distributed algorithms, distributed robotics, and algorithm design for programmable particles. In 2018, he started a postdoctoral research with Aix-Marseille University, where he worked on dynamic graphs. Currently, he is an associate professor with the Sapienza University of Rome.

Leonardo Querzoni received the PhD degree, in 2007, with a thesis on efficient data routing algorithms for publish/subscribe middleware systems. He is an associate professor with the Sapienza University of Rome. He has authored more than 80 papers published in international scientific journals and conferences. In 2016, he has coauthored the Italian National Framework for Cyber Security as a member of the Cyber Intelligence and Information Security Research Center, Sapienza University of Rome. His research interests include a range from cyber security to distributed systems and focus, in particular, on topics that include binary analysis, distributed stream processing, dependability, and security in distributed systems. In 2017, he received the Test of Time Award from the ACM International Conference on Distributed Event-Based Systems for the paper TERA: Topic-Based Event Routing for Peer-to-Peer Architectures (2007). In 2014, he was the general chair of the International Conference on Principles of Distributed Systems and was the program co-chair of the ACM International Conference on Distributed Event-Based Systems, in 2019.