Appendix: How could Neural Networks understand Programs?

Dinglan Peng [1] **Shuxin Zheng** [2] **Yatao Li** [2] **Guolin Ke** [2] **Di He** [2] **Tie-Yan Liu** [2]

A. Details of Abstract Environment Information

We develop a LLVM pass to produce the environment information. In the environment information, arguments of the functions are named $a_i, i = 0, 1, 2, ..., N_{\text{Arguments}} - 1$; SSA values are named $v_i, i = 0, 1, 2, ..., N_{\text{Values}} - 1$; stack variable allocated by *alloca* instruction are named $m_i, i = 0, 1, 2, ..., N_{\text{StackVariables}} - 1$. There may be one or more constraints of environment for every instruction, and constraints are separated by semicolon.

A.1. Constraints of Arithmetic and Logical Operations

Constraints of arithmetic and logical operations can be represented as follows: $v_i = x \text{ op } y$ or $v_i = \text{op } x$, where v_i is the SSA value of the result of the operation, x, y can be SSA values, arguments or constants, and op can be binary or unary operators such as $+, -, *, /$ or $\text{trunc}, \text{fptoint}$.

A.2. Constraints of Memory Operations

For allocating memory for stack variables, the constraint can be represented as $v_i = \text{reference } m_j$, where v_i is the address of the allocated stack variable m_j.

For memory load, the constraint can be represented as $v_i \leftarrow m_j = y_0, y_1, ...$ or $v_i \leftarrow \text{dereference } x = y_0, y_1, ...$, where v_i is loaded value and m_j is the loaded stack variable name if the memory address points to a stack variable, otherwise x is the memory address to load. $y_0, y_1, ...$ are possible values loaded from the memory address analyzed by *MemorySSA* pass.

For memory store, the constraint can be represented as $v_i \rightarrow m_j$ or $v_i \rightarrow \text{dereference } x$, where v_i is the value to store in the memory and m_j is the stack variable to be stored if the memory address points to a stack variable, otherwise x is the memory address, which can be either SSA values, arguments or constants.

For getting element pointer, the constraint can be represented as $v_i = \text{gep } x \ y_0, \ y_1, ...$, where base pointer x can be SSA values, arguments, stack variables or constants, and indices y_i can be v_i, a_i or constants.

A.3. Constraints of Selection

For PHI node, the constraint can be represented as $v_i = x_0, x_1, ...$, where $x_0, x_1, ...$ are possible values of v_i.

For selecting value by condition value, the constraint can be represented as $v_i = \text{select } x \ y_0 \ y_1$, where x is a boolean condition value, and y_0, y_1 are selected values when x is true or false respectively.

A.4. Other Constraints

For return value, the constraint can be represented as $\text{ret} = x$, which means that the return value of the function is x.

For loop depth, the constraint can be represented as $\text{loop} = 0, 1, 2, ...$, which shows the loop depth of the current instruction analyzed by *LoopInfo* pass.

B. Details of Architecture

We have two separated positional condition encoding for IR and Env.. For three kinds of IR encoding, there is a special code for `[CLS]`. And for true encoding and false encoding, there is a special code -1 for the unknown position. The unknown code is for the instructions like *switch* or *call*, for which we cannot decide the next position or there are more than two target positions. A *switch* instruction can also be converted to a sequence of branches to prevent unknown position code in this case. For Env. encoding, it is similar except that `[CLS]` is replaced by `[SEP]`. The hidden state of `[CLS]` in the last layer of the instruction-level transformer is connected to a MoCo head. The dimension of the MoCo head is 256 and the length of the MoCo queue is 65536. Finally, when applying masked language model, an IR instruction and its corresponding Env. constraints won't be masked at the same time.

[1]University of Science and Technology of China [2]Microsoft Research Asia. Correspondence to: Shuxin Zheng, Yatao Li, Di He <{shuz,yatli,dihe}@microsoft.com>.

Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

C. The influence of K

In the section of experiments, we set $K = 4$ as constant, which means that each IR and Env. Transformer encoder would process sequences with a length of 32 and 16 tokens. Larger K would lead to significant computation increment and memory consumption since the complexity of attention layers is quadratic (i.e., $\mathcal{O}(L^2)$). But in the meanwhile, larger k would also improve the capability of capturing the contextual information among long sequences. In this section, we investigate the performance gap between different choices of K in Table.1.

Table 1. **Classification error on POJ-104 test dataset.** ASTNN (Zhang et al., 2019) could access the symbol names in source code, which will be normalized in other IR-based methods.

Methods	Error(%)
ASTNN (Zhang et al., 2019)	1.8
OSCAR ($K = 4$)	1.92
OSCAR ($K = 16$)	**1.72**

D. Hardware-related Program Semantics Understanding

In this section, we investigate whether OSCAR performs well on hardware-related semantics understanding. We conduct experiments on two widely-used tasks: device mapping and coarsening threads predictions. We exactly follow the same experimental settings with (Ben-Nun et al., 2018; Cummins et al., 2020). The results have shown in Table.2 and 3. In both experiments, OSCAR performs well comparing to baseline methods, and shows good capabilities of program semantics understanding on hardware-related tasks.

Table 2. Error rate (%) on device mapping task.

	AMD	NVIDIA
DeepTune$_{IR}$	26.9	31.6
inst2vec(Ben-Nun et al., 2018)	19.7	21.5
ProGraML(Cummins et al., 2020)	13.4	20.0
OSCAR	**11.2**	**10.3**

Table 3. Speedups achieved by coarsening threads

	DeepTune$_{IR}$	inst2vec	inst2vec-imm	OSCAR
Cypress	1.17	**1.37**	1.28	1.35
Tahiti	1.23	1.10	1.18	**1.30**
Fermi	1.14	1.07	1.11	**1.27**
Kapler	0.93	1.06	1.00	**1.12**

E. Compliant Options for Constrative Learning

We totally generate 19 variants for every function from different sequences of LLVM passes. Firstly, we generate three variants using *opt* of the LLVM toolchain with standard passes of -O1/2/3. Then for every LLVM IR assembly file, we randomly drop and shuffle the passes of the -O2 optimization level and use *opt* to generate the variants. The standard -O2 optimization passes are shown in Tab.4. The algorithm for generating the passes is as follows:

Algorithm 1 Generating LLVM passes

```
Input: List of the standard -O2 optimization passes $P$, maximal shuffled items $M$ which is even.
Output: List of the generated optimization passes $P'$
1 Generate a random integer $N \in [0, \mathrm{len}(P)]$;
2 Randomly select $N$ items $P'$ from $P$;
3 Generate a random even integer $m \in [0, M]$;
4 Randomly select $m$ unique items $S$ from $\{0, 1, ..., N-1\}$;
5 for $i \leftarrow 0$ to $m-2$ by $2$ do
6     $P'[S[i]] \leftrightarrow P'[S[i+1]]$;
7 end
```

In our case, we set M = 20.

F. Pre-training Data and Pre-Processing

F.1. Pre-training Data

We assembled a large corpus of real-world programs for pre-training from publicly available open-source non-fork GitHub repositories, summarized in Table.5. The software covers a broad range of disciplines from operating systems and compilers, to machine learning systems and linear algebra subprograms. After collecting the corpus, we first compile them into LLVM IRs using Clang 11 with -O0 optimization level[1]. Then, for each program, we further generate 19 variants with the same functionality (20 in total), by random arrangement and combination of different LLVM optimization passes. After that, we sample about 500k functions from the dataset. In the pre-training phase, we sample several functions from the dataset to form a mini-batch as the training data for each iteration.

F.2. Pre-Processing

Firstly, we use wllvm[2] with Clang 11 to compile the source code to LLVM IR. For every object file, wllvm will generate an LLVM IR bitcode file, which can be then converted to

[1]Except for Linux Kernels which could only be built with -O1 or above.

[2]`https://github.com/travitch/whole-program-llvm`

an LLVM IR assembly file. For every LLVM IR assembly file, we extract the functions which occur in all 20 variants of the file.

Then we use the above-mentioned LLVM pass to filter out the functions which exceed the maximal instructions as well as generate PCE, environment information, and IR instructions with normalized identifier names.

After that, we tokenize the LLVM IR assembly code and process the names of functions and types as follows:

1. If an identifier name is a mangled C++ symbol, demangle it and remove extra information. Only function names will be retained. Also, for type names, extra information such as template arguments or namespaces will be removed.
2. Split the names into words by underscore and case.
3. Use byte pair encoding to break down the words into subwords.

Literal constants will also be split into subwords using BPE.

Finally, we convert IR instructions and environment information into raw text and split them into the training set and the validation set with the ratio of 19:1.

G. Downstream Dataset

G.1. Binary Diffing

We collected several programs and libraries. The numbers of the programs in the training/validation/testing dataset are 13, 2, and 5. Firstly, we compile them using GCC 7.5.0 with debug information and four different optimizations levels (-O0/1/2/3). Then, we use debug information to generate the ground truth of matched functions in different variants of binaries and then stripped the debug information out of the binaries as well as replace the function symbols with meaningless strings. We only treat two binary functions as equivalent if their function names and source code filenames in the debug information are both the same. In this way, we can ensure that the ground truth collected is correct, though it may not be exhaustive. After that, we use retdec[3] decompiler to convert the binaries to LLVM IR, and then process the IR to generate raw text input in the above-mentioned way.

For the training and validation set, only the functions that occur in all four variants of a binary will be used. However, for the test set, all the functions will be included as we need to retrieve a function from all the functions of another binary. The numbers of the functions in the training/validation/testing dataset are 71000, 5804, and 40791.

[3]https://retdec.com/

Before matching the functions using BinDiff(Dullien & Rolles, 2005), we remove the names of the functions in IDA except for the exported symbols as BinDiff will match two functions if they have the same name, which results in invalid results.

We use *Recall@1* as the evaluation metrics, which can be computed in this manner:

For binaries B_1, B_2 as the sets of binary functions, we have a ground truth mapping $f_1 : B_1' \to B_2'$, where $B_1' \subseteq B_1, B_2' \subseteq B_2$. For every $x_1 \in B_1'$, we also find a $x_2 = f_2(x_1) \in B_2$ which maximizes $\text{similarity}(x_1, x_2)$ computed by our model, which is the cosine similarity of the `[CLS]` feature vectors of these two functions. MoCo(He et al., 2020) head is not involved in the computation of the feature vectors. Then, we have:

$$\text{Recall@1} = \frac{|f_1 \cap f_2|}{|f_1|}$$

G.2. POJ-104

POJ-104 dataset(Mou et al., 2016) is collected from an online judge platform, which contains 104 program classes written by 500 different people randomly selected per class, so there are a total of 52000 samples in the dataset. We use the dataset for the task of clone detection and algorithm classification.

For the POJ-104 clone detection task, we compile the code of the POJ-104 dataset to LLVM IR assembly files with Clang 11 and -O0 optimization level. To compile the code successfully, we prepend following statements before the code:

```
#include <bits/stdc++.h>
using namespace std;
```

Then, we replace *void main* with *int main* and disable all the warnings to compile the source code. After that, we extract the IR instructions, environment information , and PCE information from the produced LLVM IR assembly files in the above-mentioned way. We concatenate the functions in an LLVM IR assembly file into a single input sequence and truncate it to 255 instructions.

Finally, we split the dataset according to the labels. 64 classes of programs are used for training; 16 classes of programs are used for validation; 24 classes of programs are used for testing.

For the algorithm classification task, we use the compiled IR files from the dataset processed by NCC(Ben-Nun et al., 2018)[4]. The dataset is split by 3:1:1 for training, validation,

[4]https://github.com/spcl/ncc/tree/master/task

and testing. To successfully compile the programs, *#include* statements are also prepended before the source code. Data augmentation is applied on the training set by compiling each file 8 times with different optimization options (-OO/1/2/3 and w/ or w/o -ffast-math). We keep up to four functions per source code file and truncate each function to 255 instructions.

We use MAP@R as the evaluation metrics of the clone detection task. MAP@R is defined as the mean of average precision scores, each of which is evaluated for retrieving R most similar samples given a query. In our case some source code files (˜3%) are not compilable, so we only retrieve R_i most similar samples for every query where R_i is the number of the valid samples of the same class with the query s_i. Detailed information of how to compute our evaluation metrics is as follows.

We denote the set of all the samples as $S = \{s_i \mid i = 0, 1, 2, ..., N-1\}$, where N is the number of the samples. And the label of s_i is $l(s_i)$. Then, we denote the similarity scores between s_i and s_j computed by our model f as $\text{similarity}(s_i, s_j) = \cos(f(s_i), f(s_j))$. The feature vectors $f(s_i)$ and $f(s_j)$ computed by our model are the output of the two-layer MLP of the MoCo head.

For every $s_i \in S$, let $S_i = \{s_j \in S \mid l(s_j) = l(s_i), s_j \neq s_i\}$, and $R_i = |S_i|$. We retrieve R_i most similar samples as Q_i from $S - \{s_i\}$ by similarity scores $\text{similarity}(s_i, s_j), s_j \in S - \{s_i\}$. Then, we have:

$$\text{Precision}_i = \frac{|Q_i \cap S_i|}{|S_i|}$$

$$\text{MAP@R} = \frac{1}{N} \sum_{i=0}^{N-1} \text{Precision}_i$$

H. Training Details

H.1. Pre-training

The loss for the pre-training task is:

$$L = \lambda L_{\text{MLM}} + \mu L_{\text{MoCo}}$$

where λ is MLM loss coefficient and μ is MoCo loss coefficient. We strictly follow the algorithm of MoCo, except that x_{key} is an augmented image in MoCo, while $x_{\text{key}} = [x_{\text{IRkey}} : x_{\text{EnvKey}}]$ is the augmented IR instruction and its environment information in our model.

We pretrain the model on 8 V100 GPUs with the hyperparameters shown in Tab.6.

H.2. Binary Diffing

When training OSCAR for the binary diffing task, we firstly sample a mini-batch of triplets of two samples $v_i, v_i^+ (i = 0, 1, 2, ..., N-1$,$N$ is the size of the mini-batch) of the same label, i.e. from the binary functions generated by different optimizations with the same source code function, and one sample of another label $v_i^- (i = 0, 1, ..., N-1)$. The feature vectors of the triplets are denoted $v_0, v_0^+, v_0^-; v_1, v_1^+, v_1^-; ...; v_{N-1}, v_{N-1}^+, v_{N-1}^-$. The label of v_i is $l(v_i)$, and we have $l(v_i) = l(v_i^+) \neq l(v_i^-)$. The loss L of the mini-batch is computed as follows:

$$\tau = \sqrt{d} = \sqrt{768}$$

$$p_i = \exp(v_i \cdot v_i^+ / \tau)$$

$$s_{ij} = \exp(v_i \cdot v_j / \tau) + \exp(v_i \cdot v_j^+ / \tau)$$

$$n_i = \exp(v_i \cdot v_i^- / \tau) + \sum_{j=0;\ l(v_j) \neq l(v_i)}^{N-1} s_{ij}$$

$$L = -\frac{1}{N} \sum_{i=0}^{N-1} \log \frac{p_i}{p_i + n_i}$$

The feature vectors are the last-layer hidden states of the `[CLS]` tokens in the instruction-level transformer. MoCo head including the two-layer MLP is dropped. We train the model on 4 V100 GPUs for 128000 steps with 6400 warm-up steps. Peak learning rate is 0.00002; weight decay is 0.2; dropout and attention dropout is 0.1; batch size is 48 and update frequency is 1.

We use BinDiff, Asm2vec(Ding et al., 2019)[5] and BinaryAI(Yu et al., 2020a;b)[6] v2 API as the baseline. All hyperparameters of Asm2vec are default. BinaryAI uses IDA Pro and its Hex-Rays decompiler to generate C-like pseudo-code for binary functions, and then upload it to Tencent's server to compute the similarity of the functions. Also, Asm2vec and BinDiff both depend on IDA Pro and its dissembler or decompiler. As the Hex-Rays decompiler is considered better than the retdec decompiler, we think that the comparison between OSCAR and BinaryAI is reasonable.

H.3. Code Classification

We firstly sum the `[CLS]` vectors of each function in an LLVM IR assembly file to get the representation of the sample. Then the feature vectors are feed into a fully connected layer followed by a projection layer and a softmax layer. After that, we use the cross-entropy loss for the classification task.

[5] https://github.com/McGill-DMaS/Kam1n0-Community

[6] https://github.com/binaryai/sdk

We train the model on 8 V100 GPUs for 100000 steps with 10000 warm-up steps. Peak learning rate is 0.00005; weight decay is 0.01; dropout and attention dropout is 0.1; batch size is 8 and update frequency is 4.

References

Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems*, 31:3585–3597, 2018.

Cummins, C., Leather, H., Fisches, Z., Ben-Nun, T., Hoefler, T., and O'Boyle, M. Deep data flow analysis. *arXiv preprint arXiv:2012.01470*, 2020.

Ding, S. H., Fung, B. C., and Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 472–489. IEEE, 2019.

Dullien, T. and Rolles, R. Graph-based comparison of executable objects (english version). *Sstic*, 5(1):3, 2005.

He, K., Fan, H., Wu, Y., Xie, S., and Girshick, R. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9729–9738, 2020.

Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., and Wu, S. Order matters: semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 1145–1152, 2020a.

Yu, Z., Zheng, W., Wang, J., Tang, Q., Nie, S., and Wu, S. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33, 2020b.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794. IEEE, 2019.

Table 4. -O2 optimization passes

-tbaa	-scoped-noalias
-forceattrs	-inferattrs
-ipsccp	-called-value-propagation
-attributor	-globalopt
-mem2reg	-deadargelim
-instcombine	-simplifycfg
-prune-eh	-functionattrs
-sroa	-early-cse-memssa
-speculative-execution	-jump-threading
-correlated-propagation	-simplifycfg
-domtree	-instcombine
-libcalls-shrinkwrap	-pgo-memop-opt
-tailcallelim	-simplifycfg
-reassociate	-loop-simplify
-lcssa	-scalar-evolution
-loop-rotate	-licm
-loop-unswitch	-simplifycfg
-instcombine	-loop-simplify
-lcssa	-scalar-evolution
-indvars	-loop-idiom
-loop-deletion	-loop-unroll
-mldst-motion	-phi-values
-gvn	-phi-values
-memcpyopt	-sccp
-demanded-bits	-bdce
-instcombine	-jump-threading
-correlated-propagation	-phi-values
-dse	-loop-simplify
-lcssa	-scalar-evolution
-licm	-adce
-simplifycfg	-instcombine
-barrier	-elim-avail-extern
-rpo-functionattrs	-globalopt
-globaldce	-float2int
-lower-constant-intrinsics	-loop-simplify
-lcssa	-scalar-evolution
-loop-rotate	-loop-distribute
-scalar-evolution	-demanded-bits
-loop-vectorize	-loop-simplify
-scalar-evolution	-loop-load-elim
-instcombine	-simplifycfg
-scalar-evolution	-demanded-bits
-slp-vectorizer	-instcombine
-loop-simplify	-lcssa
-scalar-evolution	-loop-unroll
-instcombine	-loop-simplify
-lcssa	-scalar-evolution
-licm	-transform-warning
-alignment-from-assumptions	-strip-dead-prototypes
-globaldce	-constmerge
-loop-simplify	-lcssa
-scalar-evolution	-loop-sink
-instsimplify	-div-rem-pairs
-simplifycfg	

Table 5. The eleven sources of LLVM IR used to produce pre-training dataset. All software is downloaded from Github.

Software	Domain	#instructions	#functions
Linux-vmlinux	Linux Kernel	2,930,372	45,368
Linux-modules	Linux Kernel	16,509,892	229,942
GCC	Compiler	1,816,782	22,383
MPlayer	Multimedia	1,223,068	12,747
OpenBLAS	BLAS	515,985	5,415
PostgreSQL	Database	939,199	12,807
Apache	Web Server	390,135	5,519
Blender	3-D Creation	5,925,801	123,689
ImageMagcick	Image Processing	440,265	7,182
Tensorflow	Machine Learning	12,041,852	294,553
Firefox	Browser	5,290,430	96,187
Total		48,023,781	855,792

Table 6. Hyper-parameters for pre-training.

Hyper-parameter	Value
Training steps	1000000
Warm-up steps	30000
Peak LR	0.0001
Batch size	16
Update frequency	4
Dropout	0.1
Attention dropout	0.1
Weight decay	0.01
MoCo dimension	256
MoCo temperature	0.02
MoCo momentum	0.999
MoCo queue length	65536
MLM loss coefficient	1
MoCo loss coefficient	1000