Nimbus: Toward Speed Up Function Signature Recovery via Input Resizing and Multi-Task Learning

Yi Qian[1,2], Ligeng Chen[1,2,*], Yuyang Wang[1,2], and Bing Mao[1,2]
[1]Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[2]Department of Computer Science and Technology, Nanjing University, Nanjing, China
{yi_qian, chenlg, y.y.wang}@smail.nju.edu.cn, maobing@nju.edu.cn
*corresponding author

2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-7704-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/QRS57517.2022.00053

***Abstract*—Function signature recovery is important for many binary analysis tasks such as control-flow integrity enforcement, clone detection, and bug finding. Existing works try to substitute learning-based methods with rule-based methods to reduce human effort.They made considerable efforts to enhance the system's performance, which also bring the side effect of higher resource consumption. However, recovering the function signature is more about providing information for subsequent tasks, and both efficiency and performance are significant.**

In this paper, we first propose a method called *Nimbus* for efficient function signature recovery that furthest reduces the whole-process resource consumption without performance loss. Thanks to information bias and task relation (i.e., the relation between parameter count and parameter type recovery), we utilize selective inputs and introduce multi-task learning (MTL) structure for function signature recovery to reduce computational resource consumption, and fully leverage mutual information. Our experimental results show that, with only about the one-eighth processing time of the state-of-the-art method, we even achieve about 1% more prediction accuracy over all function signature recovery tasks.

Keywords*—*Function signature; multi-task learning; recurrent neural network

I. Introduction

Function signature recovery plays an important role in binary analysis, widely used in many security analysis works as pre-processing such as bug finding [1], [2], clone detection [3], [4], code hardening [5]–[11], etc. It is composed of two tasks, parameter count recovery, and parameter type recovery.

Existing Works. Function signature recovery is challenging work since most binaries in real applications are stripped, which lose almost all high-level semantics and retain only low-level information via machine code.

The majority of previous works mainly rely on experienced analysts to recover the missing semantics from binary code [9], [10], [12]–[16]. Recently, some researchers leverage machine learning-based methods to avoid extracting excessive rules via much human effort. EKLAVYA [17] and ReSIL [18] utilize gated recurrent unit (abbreviated as GRU) [19] (one kind of recurrent neural network) and get surprising results. Coincidentally, both of them focus on improving recovery performance (i.e., accuracy), but they leave the problem of *resource consumption* aside, which should be taken into consideration for production.

Our Solution. In this work, we take all the advantages of machine learning models and try to optimize resource consumption from the whole lifecycle of tool construction. We introduce the 2 key designs to construct our efficient function signature recovery tool *Nimbus*[1] as follows, i.e., input reduction and multi-task learning.

❶***Reduce the input size via information bias.*** According to our empirical study of a considerable dataset, we find that the information about function signature is mainly gathered in the front of a function rather than uniformly distributed throughout the function (no matter in binary or assembly level). Taking the input from the function head achieves more precise and faster performance growth in all function signature recovery tasks than inputting it all into the procedure.

❷***Merging the learning models via mutual information of different tasks.*** Intuitively, the data distribution of parameter count and parameter type are relevant to each other as well as the function semantics. According to our data analysis, distribution relations widely exist. Existing works treat each function signature recovery task independently. Specifically, they recover each function signature (amount or type) with an independent model, called single-task learning (STL) structure. This not only requires independent models to perform prediction tasks separately, consuming a lot of memory and running time but also ignores the mutual information between task associations.

So we introduce multi-task learning (MTL) [20] structure, which enables deep learning models to train on multiple related tasks on one model and eventually get multiple outputs for different tasks. MTL avoids repetitive work compared with the STL structure, saving resources in both training and testing procedures. In addition, the MTL structure fully utilizes the related information via recovery tasks, improving the performance and the ability to generalization.

Evaluation. We set up a credible dataset following the previous works, and thoroughly evaluate our system *Nimbus* on different aspects. Compared with the state-of-the-art method EKLAVYA, *Nimbus* is **9.92**× faster on the training procedure and saves about **87.8%** time in the prediction procedure both on GPU-equipped and CPU-only hardware environments. Our lightweight and efficient tool design can help security analysts perform pre-analysis or provide analysis results for downstream tasks.

[1]The name of our system is derived from the book *"Harry Potter"*, and the name is taken from the main character's broomstick *Nimbus 2000*.

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00053

The contributions of this paper are as follows:

- According to the empirical study, we verify the information bias phenomenon in each binary level function, and we prune the functions to keep highly informative instructions as input.
- According to the intuitive relations of sub-tasks in function signature recovery, we first introduce the multi-task learning structure to enhance function signature recovery and customize an MTL-GRU architecture.
- We evaluate the prototype of *Nimbus* on the dataset, and our work achieves a significant reduction in resource consumption, while even getting about a 1% accuracy increase.

The rest of the paper is organized as follows, Section II introduces the motivation of our work. Section III defines the problem. Section IV presents the workflow and our system design. Section V evaluates our model. Section VI presents the discussion. Section VII discusses some related works and Section VIII concludes the paper.

II. MOTIVATION

Most learning-based function signature recovery works are hard to use as a tool because they require hours or even days of training on high-power GPUs and cost a lot when predicting even though they achieve good performance. For example, we rebuild EKLAVYA and find that the average time for predicting a single function using GPU and CPU is **17.88ms** and **70.40ms**, respectively. As a pre-work for many binary analysis tasks, such resource consumption may affect their efficiency, especially without high-power GPUs. In this paper, we try to reduce the resource consumption of both training and prediction. Nowadays, most works focus on improving model performance, our work balance the performance and overhead. We hope this contributes to the community.

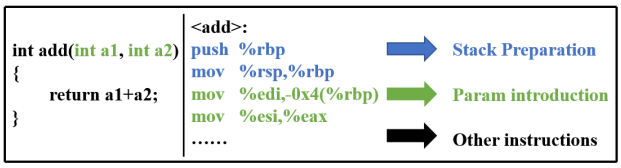


Figure 1: Source code and its assembly code.

A. Model input

Most existing works use unfixed-length assembly codes or byte codes as model inputs. Obviously, the longer the input length, the more information the model obtains, and the easier it is to improve the model performance. However, longer inputs also require more computational resources. By reducing the input length, we can save computational resources from the source. By following the data selection mechanism of EKLAVYA, we compile some widely used open-source projects into binaries as our dataset. We analyze the dataset and there are two key findings in the selection of model inputs. The detail of the dataset is given in Section V.

*1) **Information bias - Does the information in the content of the code follows a uniform distribution**:* During analysis, we find that although the use of parameters is scattered through the whole function, they are often introduced at the function head. Figure 1 gives an example function and its assembly code, and the import of parameters clusters at the function head.

We anticipate that this phenomenon widely exists in binary codes, i.e., that information about the function signature clusters at the function head and we call this *information bias*. We do several experiments to verify the existence of information bias in Section V.

*2) **Input length - What input length is the most appropriate**:* According to the analysis results of the function length shown in Figure 2, we find that about 30% of the functions are less than 40 instructions and about 70% of the functions are less than 120 instructions, i.e., most functions are not that long. Due to information bias, when the input reaches a certain length, the following instruction may contain little valuable information, so the overlong inputs may not be required.

Another interesting finding is a large gap between the mean and the median of the function lengths. Precisely, the mean is 165 and the median is 74. Combined with the statistical distribution, we believe that the median is more representative of the general function length. To observe the effect of different input lengths on the model, we train models with different input lengths around the median in Section V.

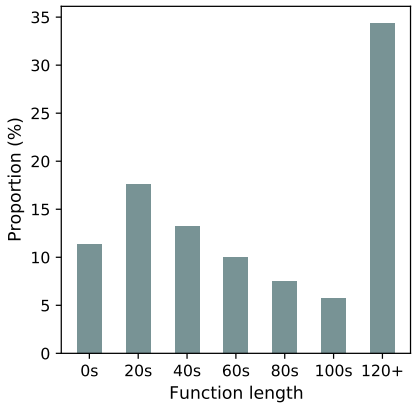


Figure 2: Function lengths and their proportions. '0s' represents the function length is between [0, 20), and so on.

B. Model structure

Previous works treat function signature recovery tasks as independent and use STL models to recover them. Specifically, one model can only recover parameter count or one parameter type at one specific location, so multiple models are required to accomplish function signature recovery. However, in different models, the same layers undertake often similar work as these tasks are related, which undoubtedly leads to a waste of computational resources. As a result, in the case of limited computational power, it takes multiple times to train and use models. If we can avoid repeated computation in those STL

models, we can reduce computational resource consumption structurally.

*1) **Task relation**:* Function signature recovery consists of two parts: parameter count recovery and parameter type recovery. Intuitively, there are two possible relations between tasks. One is the relation between parameter count and parameter types, and the other is between two parameter types in different positions. We try to explore if these relations really exist. And if so, whether we could improve performance with them.

To verify the intuition, we analyze the dataset, and the results are shown in Figure 3. According to the results, the parameter count and the parameter type are correlated, and so do the parameter types in different positions. For example, when the first parameter is *struct**, the second one is most probably *struct** as well. In fact, we find that the above relations widely exist in the dataset.

*2) **Multi-task learning structure**:* The MTL structure [20] is proposed to make the model learn the information between related tasks. It consists of shared layers and task-specific layers. Different tasks share their intermediate representation at the shared layer and get task-specific results at the task-specific layer.

Nimbus benefits a lot from the MTL structure. First, it helps *Nimbus* to select features. For a given task, its related tasks give *Nimbus* the evidence of which features are useful and help *Nimbus* to focus more on them. Second, the MTL helps *Nimbus* learns more generalized representations. A model will be overfitting if it learns both data and noise during training. MTL forces *Nimbus* to adapt to the noise of different tasks, which reduces overfitting and makes *Nimbus* more generalized. Last but not the least, MTL actually merges multiple task-specific layers into one shared layer and it obviously avoids repeated computation thus reducing resource consumption.

III. PROBLEM DEFINITION

The distribution of parameter counts and parameter types are shown in Figure 4 (a) and Figure 4 (b), respectively. Our function signature recovery tasks are defined as follows.

- **Parameter Count**: The number of parameter passed to function, **abbreviated as** PC.

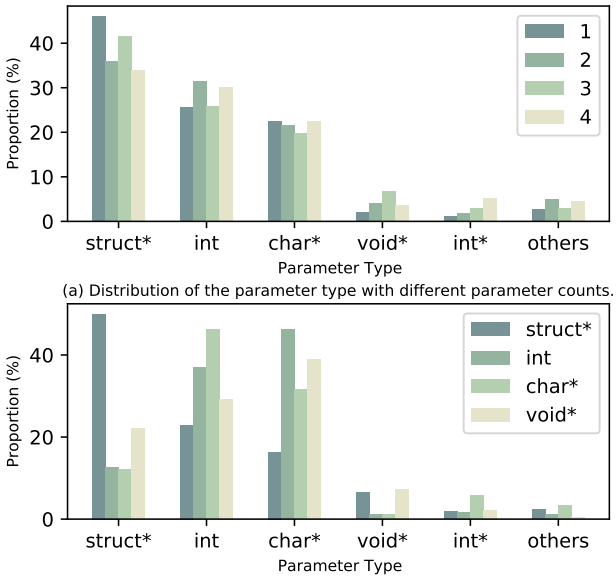
$$PC \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, others\} \tag{1}$$

- **Parameter Type**: Parameter type for each parameter passed to function, **abbreviated as** PT. PT_i represents the parameter at the i_{th} position $(i = 1, 2, ...)$.

$$\begin{aligned} PT \in \{&struct*, int, char*, void*, int*, enum, \\ &char, void, float, struct, others, NULL\}, \end{aligned} \tag{2}$$

where *NULL* denotes that the position has no parameters.

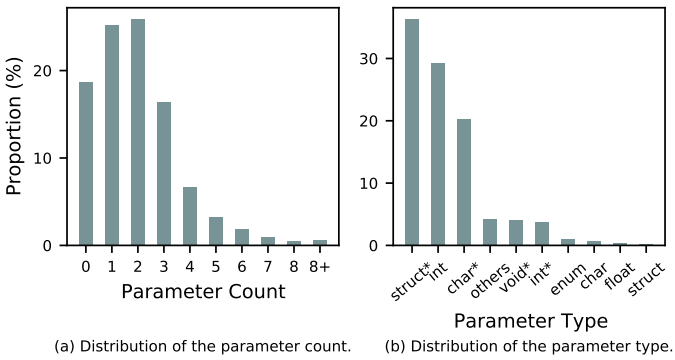
We make such definitions because over 99% of the function parameters are less than 9, and parameter types defined in PT (except *others* and *NULL*) account for more than 95% of all parameter types. Note that our PT is different from EKLAVYA, we remove the *union* because we think the *union* has more high-level semantics. In assembly code, *union* will be translated into a certain parameter type. In addition, we add other parameter types such as *char** to make our model suitable for more situations.



(a) Distribution of the parameter type with different parameter counts.

(b) Relationship between the first parameter type and the second parameter type.

Figure 3: Relations of recover targets. To make the relations clearer we have omitted some types that have a smaller proportion.



(a) Distribution of the parameter count. (b) Distribution of the parameter type.

Figure 4: Proportion of parameter count and parameter type. '8+' denotes more than 8.

The model input is the assembly code from the function that can be easily obtained from the disassemblers, and the output is the function signature PC and PT defined above.

IV. DESIGN

The workflow is shown in Figure 5. There are two significant parts to our workflow. One is to vectorize the input (word embedding), and the other is to train the classification model (signature recovery). We make vectorization a separate module because we believe that the mature word embedding techniques retain more semantic information. We take assembly code as input, and we employ the MTL structure which allows the model outputs PC and multiple PT at the same time.

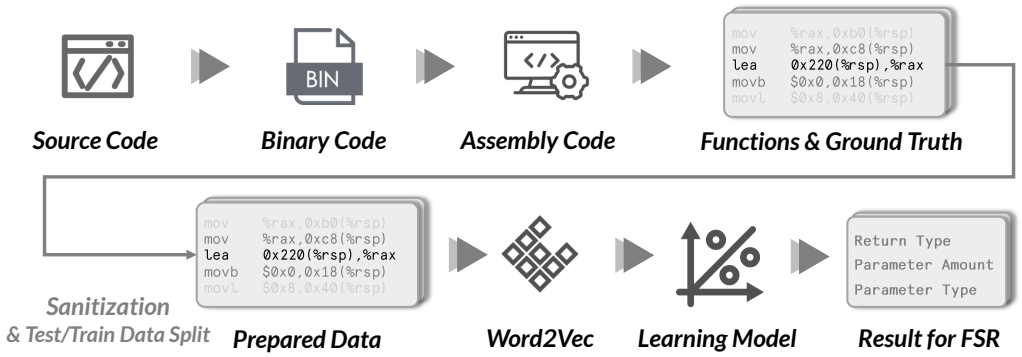


Figure 5: An overview of the steps from building a dataset to recovering function signature.

A. Word embedding

To make the input learnable for the model, the first step is to vectorize the input, which is called word embedding. By representing semantically similar text with similar high-dimensional vectors, word embedding essentially reprocesses the input and improves the representation ability.

There are many word embedding techniques, including *one-hot representation*, *word2vec* [21], and *fasttext* [22]. Our experiment employs *word2vec*, which is a mature word embedding tool that vectorizes words quickly and effectively given corpus.

Notice that we are word embedding the instruction words (mnemonics and operands) instead of the instructions. For example, we split *mov a,b* as *mov*, *a* and *b* and embedding them. Embedding instruction words has two advantages. First, it disperses the semantic information from a single vector to multiple vectors, and the dimension of a single vector can be reduced. Thus the computational consumption can be reduced. The second is that the relations within the instructions can be captured, allowing the model to learn more content details. In particular, we always split one instruction into four instruction words and truncate for larger ones because we find that most instructions can be converted into less than four instruction words, which also facilitates our subsequent processing. At last, we use the continuous bag of words (CBOW) negative sampling method in *word2vec* to train instruction words into 128-dimensional vectors.

B. Signature recovery

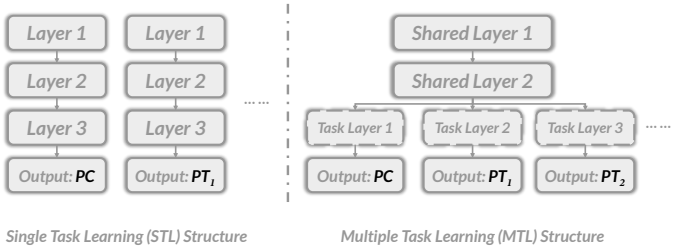


Figure 6: Model structure for recovering PC, PT_1, PT_2, PT_3, the model has four outputs.

As we discussed in Section III, most functions contain three or fewer parameters, so we employ an MTL model with one input and four outputs (denoted as PC, PT_1, PT_2 and PT_3). Our model consists of two shared layers and 4 task-specific layers, the model structure is shown in Figure 6. If we want to recover extra parameter types, we only need to add a new task-specified branch rather than a new model.

We choose to use a recurrent neural network (RNN) as the *Nimbus* network architecture. RNN introduces "memory" to the model. The network will memorize the previous information and apply it to the calculation of the current output. The "recurrent" comes from the fact that each node performs the same task. RNN is very effective for sequence data, as it mines time-series information and semantic information in the data. To alleviate the problems of gradient disappearance and gradient explosion, long-short term memory (LSTM) [23], a variant of RNN, uses forget gate and input gate to update the previously saved information. And GRU further simplifies the gate structure, merges the forget gate and the input gate into an update gate. We use GRU because it has fewer parameters and trains faster.

V. Evaluation

In this section, we try to answer the following questions, by evaluating with control variables:

- **RQ1:** How do the *different input lengths* and *resizing strategies* (from head/tail) affect the method performance?
- **RQ2:** How does the *network structure* (MTL/STL) affect the method performance?
- **RQ3:** How much does the optimization of the whole process save *resource consumption*?

Our experiments are performed on a server containing one 12-core Ryzen 3900X CPU with 48GB of RAM, and one GeForce RTX 3080 GPU with 10GB of memory. The neural network and data processing routines are written in Python, using Keras [24].

A. Dataset

We set up a dataset for method evaluation. The binaries are compiled from source files often used in the community, listed in Table I.

1) Function Extraction: We compile the source files under different configurations, and the final dataset consists of binary files compiled with different compilers and versions (*clang 3.0, clang 4.0, clang 5.0, clang 6.0, clang 7, gcc 5, gcc 6, gcc 7, gcc 8, gcc 9*), and different optimization levels (*-O0, -O1, -O2, -O3, -Os*) for x64. We use *objdump* [25] to disassemble the binary code and get their assembly code in AT&T format. Finally, we get a dataset consisting of 2,819,495 functions.

TABLE I: Project and version of dataset.

Project	Version	Project	Version
Coreutils	8.31	Gtypist	2.9.5
Inetutils	1.9.4	Binutils	2.32
Grep	3.3	Gawk	5.0.0
Nginix	1.15.12	Sed	4.7
Libpng	1.6.37	Bash	5.0
Cflow	-	Less	530
BC	1.07	Bison	3.4
Nano	4.4	Indent	2.2.12
Wget	1.20.3	Gzip	1.9

2) Sanitization and duplication: To avoid the interference of irrelevant information, we sanitize the specific address and function name. Listing 1 shows a brief example. As the instructions for direct jump, we replace the concrete address with '*IMM*' (e.g., row 1, 3). As the function reference, we replace the concrete function name with '*FUNC*' (e.g., row 2). To avoid repeated functions, we filter the functions in the dataset based on MD5 after sanitization, and only one repeated function is retained to ensure data balance. Only 272,900 distinct functions are retained after sanitization and duplication.

Through compilation, we find that binaries compiled from different projects and optimization levels may the same. To avoid information leaks, sanitization only randomly keeps one of them, which makes the evaluation of different optimization levels biased, so we do not distinguish optimization levels in the following discussion.

```
1 mov 0x2063a3(%rip), %rsi
2 je  401f31<add+0x34>
3 movabs $0xaaaaaaaa9, %rax
4 cmp %rax,%rsi
```

```
1 mov IMM(%rip),%rsi
2 je IMM<FUNC>
3 movabs IMM,%rax
4 cmp %rax,%rsi
```

Listing 1: Assembly code before and after sanitization

We obtain the ground truth for the function signature by analyzing the *DWARF debugging information* [26]. We divide the dataset into a training set and a testing set at 8:2.

B. Metrics

1) Performance evaluation: Precision (*P*) and recall (*R*) are commonly used to evaluate the model performance, which are calculated as

$$P = \frac{TP}{TP + FP}; R = \frac{TP}{TP + FN}, \tag{3}$$

where *TP*, *FP*, and *FN* denote true positive, false positive, and false negative, respectively.

To measure the performance, we use weighted accuracy Acc which can be calculated as

$$Acc = \sum_{i=1}^{n} \sigma_i \times R_i, \tag{4}$$

where n denotes the number of classes and σ_i denotes the proportion of the label i in the test set.

Weighted accuracy represents the correctly predicted rate in the test set which is a widely used metric for multi-classification tasks such as emotion recognition [27]–[29], malware classification [30] and text classification [31]. Since weighted accuracy reveals the model's global performance and emphasizes the effect of every label simultaneously, EKLAVYA uses it and so do we.

2) Resource consumption evaluation: Since the computational resource is valuable, we also take optimizing the resource consumption into account besides the model performance. We define *Efficiency*:

$$Efficiency = \frac{\sum_{k=1}^{N_t} Acc_k}{\sum_{i=1}^{N_G} T_i \times U_i}, \tag{5}$$

where N_t denotes task number, N_G denotes GPU number, T_i denotes time consumption and U_i denotes GPU usage percentage, respectively. The larger of efficiency, the smaller the resource consumed to reach the same accuracy relatively.

C. Experiment on performance

*1) **Compared with EKLAVYA**:* We compare *Nimbus* with EKLAVYA in our dataset. We use cross-entropy loss for our classification tasks, and the optimizer is *Adam* [32] with the learning rate of 1e-4, $\beta_1 = 0.9$, $\beta_2 = 0.999$ like many previous works. We use a dropout [33] probability of 0.2 on all layers to alleviate overfitting. We use 128-dimensional vectors embedded from 40 instructions as input and train for 100 epochs. To make the evaluation fair enough, we reproduce EKLAVYA referring to their paper, with the same hyper-parameters, the results are shown in Table II. *Nimbus* performs slightly better than EKLAVYA, we achieve about 1% more prediction accuracy over all function signature recovery tasks. Column 2 (i.e., *PC*) represents the performance of recovering the number of function parameters. Column 3 to 5 (i.e., PT_1, PT_2, PT_3) represent the performance of recovering variable types of different position parameters, respectively.

TABLE II: Model accuracy comparison.

Method	PC (%)	PT_1 (%)	PT_2 (%)	PT_3 (%)
EKLAVYA	96.42	94.88	95.40	97.88
Nimbus	**97.25 (+0.83)**	**95.88 (+1.00)**	**96.82 (+1.42)**	**98.40 (+0.52)**

We further do experiments on inputs and model structures of our model. The settings remain the same if not mentioned. We find that MTL benefits our model in performance with appropriate input according to the later experiment.

*2) **Information bias shortens the input length**:* We train an STL-GRU model with different instruction lengths and positions, and the results are shown in Table III. The size represents how many instructions are used as input. The location represents where the instructions come from.

TABLE III: Ablation study of different locations and different instructions' input lengths. (Accuracy)

Size	Location	PC (%)	PT_1 (%)	PT_2 (%)	PT_3 (%)
5	Head	57.97	65.22	62.02	75.12
	Tail	47.43	57.25	55.13	74.69
10	Head	89.78	85.79	86.68	91.87
	Tail	66.17	75.02	72.41	82.74
20	Head	96.23	94.23	95.15	97.25
	Tail	81.68	87.04	85.57	90.28
40	Head	**96.74**	**95.46**	**96.23**	**97.76**
	Tail	87.99	90.99	90.45	93.52
80	Head	96.72	95.84	96.35	97.71
	Tail	92.06	93.38	93.62	95.67
120	Head	97.01	95.94	96.60	97.95
	Tail	93.86	94.63	95.00	96.50

When the size is 40 and the input comes from the head and the tail, the accuracy of PC is 96.74% and 87.99%, respectively. Experiments also show that 20 instructions from the head achieve an approximately equal performance of 120 from the tail. Actually, instructions from the head always make the model better than that from the tail in any size and any task, which verifies our intuition of information bias. In addition, the model performance becomes better as the input length increases. However, there is little accuracy growth since size reaches 40 when the input comes from function head, where the Acc of PC, PT_1, PT_2 and PT_3 are 96.74%, 95.46%, 96.23%, and 97.76% respectively. This phenomenon is caused by information bias as well, i.e., little valuable information is contained after the first 40 instructions.

In addition, with the size increase, the accuracy difference between the head and tail decreases. A reasonable explanation is that the distribution of function length cause it. As we mentioned in Section II, about 70% of functions are less than 120 instructions. In another word, when the size is 120, the instructions in the head are the same as those in the tail for about 70% of the functions. When the size is large enough, the information obtained from the tail comes from the head, which brings us back to information bias.

To sum up, we think it is best to set the model input as 40 instructions from the head to achieve satisfying results.

*3) **MTL structure makes models obtain positive information gain**:* We train STL-GRU and MTL-GRU models with different sizes shown in Table IV. All the input comes from the function head. When the size is large enough, MTL performs better than STL in all tasks, e.g., the accuracy of PC is 97.25% and 96.74% of MTL and STL with size 40. This demonstrates the relations between signature recovery tasks, and MTL models obtain information gain and improve generalization to perform better. On the contrary, MTL does not perform as well as STL when the size is insufficient, e.g., 20. We conjecture two reasons for this situation. One is information lack, and the other is noise propagation. In case of information lack, STL, focusing on a single task with limited information, is better. Besides, MTL propagates and amplifies the noise between different tasks. Combined with former experiments, *Nimbus* adopts the MTL structure.

TABLE IV: Ablation study of model structure. (Accuracy)

Size	Structure	PC (%)	PT_1(%)	PT_2 (%)	PT_3 (%)
5	MTL	57.59 ↓	65.10 ↓	62.08 ↓	75.00 ↓
	STL	57.97	65.22	62.02	75.12
10	MTL	89.53 ↓	85.01 ↓	86.10 ↓	91.34 ↓
	STL	89.78	85.79	86.68	91.87
20	MTL	95.98 ↓	94.11 ↓	95.44 ↑	97.10 ↓
	STL	96.23	94.23	95.15	97.25
40	MTL	97.25 ↑	95.87 ↑	96.82 ↑	98.46 ↑
	STL	96.74	95.46	96.23	97.76
80	MTL	97.29 ↑	96.30 ↑	97.18 ↑	98.60 ↑
	STL	96.72	95.84	96.35	97.71
120	MTL	97.12 ↑	96.34 ↑	96.88 ↑	98.14 ↑
	STL	97.01	95.94	96.60	97.95

D. Experiment on resource consumption

*1) **Compared with EKLAVYA**:* We compare *Nimbus* with EKLAVYA as shown in Table V. About embedding, EKLAVYA uses 500 instructions as input compared with our 40 instructions (160 instruction words), and the vector is 256-dimensional compared with our 128-dimensional. Training represents the average time in seconds that the model takes to train one epoch. GT and CT represents the testing time on GPU and CPU for one function in milliseconds, respectively. Theoretically, the input matrix size of EKLAVYA is about **6**× than *Nimbus*. With the MTL structure, *Nimbus* further improves efficiency. As a result, EKLAVYA is about **9.92**× longer than *Nimbus* in training time and **8**× longer both on GPU and CPU in testing time.

We change embedding and size to further explore their effect on time consumption. Even with the same input as *Nimbus*, EKLAVYA still takes **1.56**× in training and **2.96**× in testing on GPU. With the same size, our embedding method leads to more time-consuming due to the use of instruction words, but our model also performs better when using instruction words.

TABLE V: Time consuming comparison. *GT/CT* denotes testing time on GPU/CPU.

Structure	Embedding	Size	Training (s)	GT (ms)	CT (ms)
STL	EKLAVYA	500	942.01	17.88	70.40
STL	EKLAVYA	40	132.72	6.56	10.26
STL	*Nimbus*	40	216.05	7.87	24.52
MTL	***Nimbus***	**40**	**94.84 (9.9×)**	**2.20 (8.1×)**	**8.39 (8.4×)**

The following experiments indicate that MTL saves resource consumption in all aspects.

*2) **Time consumption & Efficiency**:* We record the time consumption of the previous experiments and calculate their

efficiency, shown in Table VI. *Time* represents the average time in seconds that the model takes to train one epoch. *GPU* represents the average GPU usage percent during training.

TABLE VI: Model efficiency comparison.

Size	Structure	Time (s)	GPU (%)	Efficiency (%)
5	MTL	37.09	40.52	17.28 (2.09×)
	STL	122.27	25.75	8.27
10	MTL	40.19	58.73	14.91 (1.96×)
	STL	132.82	35.00	7.62
20	MTL	47.81	79.37	10.08 (2.31×)
	STL	139.09	62.90	4.37
40	MTL	94.84	78.58	5.21 (2.08×)
	STL	216.05	71.25	2.51
80	MTL	168.43	82.73	2.79 (2.10×)
	STL	380.44	76.63	1.33

Obviously, the smaller size makes the training time shorter because it reduces the computational effort from the source. In addition, the MTL model takes less training time compared to STL models of the same size. Furthermore, the time consumption of MTL increases less compared with STL as the size increases. For example, when the size increases from 20 to 40, STL models need extra 76.96 minutes to train one epoch, but the MTL model only needs extra 47.03 minutes. The above time-saving effect is attributed to the fact that MTL merges the task-specific layers of multiple STL models into the shared layer, thus avoiding duplicate computations. We find a similar phenomenon in efficiency, indicating that MTL also makes higher utilization of computational resources.

TABLE VII: Model test time on both GPU and CPU.

Size	Structure	GPU (ms)	CPU (ms)
5	MTL	1.47	2.18
	STL	5.64	7.56
10	MTL	1.56	3.02
	STL	5.89	10.12
20	MTL	1.78	4.84
	STL	6.58	15.08
40	MTL	2.20	8.39
	STL	7.87	24.52
80	MTL	2.93	15.23
	STL	9.67	42.64

The trained model needs to be used on different machines, which most likely do not have GPUs, so we use both GPU and CPU to invoke the model to make predictions on the test set and record the run time shown in Table VII. Here, GPU and CPU represent the average time spent predicting one function signature in milliseconds.

As we can see, similar to the training, it takes less time testing with smaller input, and the time-saving effect of MTL also applies in testing both on CPU and GPU, so MTL is a CPU-friendly structure compared with STL. One important finding is that as the size increases (take MTL as an instance, from size 5 to 80), the time consumption of testing on the CPU (6.98×) grows faster than that of the GPU (1.99×), which further motivates us to use a shorter input to help CPU-only analysts improve efficiency.

*3) **Storage consumption**:* MTL reduces the number of models required to accomplish signature recovery, thus reducing the physical size that models require. In our experiment, MTL model only requires **32**MB while STL models require **60**MB in total.

VI. DISCUSSION

A. Representative work

In the above, we compare the most relevant work EKLAVYA to our work. ReSIL is a function signature recovery system optimized for EKLAVYA, we will discuss the differences between ReSIL and *Nimbus* in the following paper.

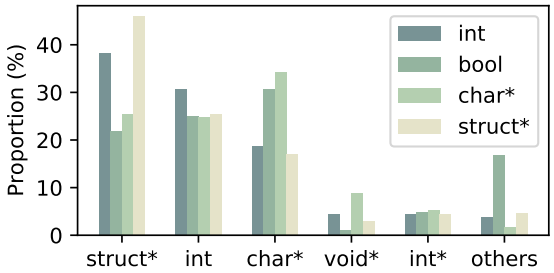


Figure 7: Distribution of parameter types under different return types.

EKLAVYA's accuracy decreases when the inputs are optimized functions. ReSIL systematically discusses the reasons for the performance degradation and optimizes it. ReSIL improves the accuracy in inferring function signatures, for example, ReSIL improved the accuracy of recovery PC from 84.8% to 92.67% at optimization level O1.

ReSIL improves accuracy by essentially using domain-specific knowledge. Inserting additional instructions to the input actually provides extra information from human knowledge, and removing unrelated instructions helps the model to focus more on useful features.

TABLE VIII: The effect of different locations and input length of the instructions on the model accuracy.

Size	Location	RT (%)
10	Head	67
	Tail	72
20	Head	73
	Tail	75
40	Head	74
	Tail	76

Different from ReSIL, our work improves not only performance but also efficiency. We prefer to make the model more usable without introducing domain-specific knowledge. At the same time, ReSIL and our improvement approach are compatible, which means that both works can be applied simultaneously.

B. Return type

The return type, as one of the characteristics of the function, can also provide part of the pre-information for function signature recovery. During exploratory data analysis, we find that the information of function return type also has spatial locality, clustered at the function tail. To verify the intuition, we do similar ablation experiments on return type and the results are shown in Table VIII.

The result indicates that the instructions from the bottom of a function can provide more related information about the function return type.

In addition, we also find differences in the distribution of parameter types under different return types, as shown in Figure 7. Obviously, functions with the return type of *struct** employ more than 40% of the parameters with type *struct**. The aforementioned information bias and distribution preference of return type may help improve function signature recovery or other security tasks in the future.

VII. RELATED WORKS

A. Information recovery from stripped binaries

Parameter type recovery in function signature recovery is essentially the process of recovering high-level semantic information (variable types) from stripped binaries, which can be seen as a special kind of task of type inference. Type inference can be basically divided into two categories, rule-based and machine learning-based.

Lin et al. [14] formulate rules based on expert knowledge for type inference. [13], DIVINE [12], TIE [15], SecondWrite [16] also use analysis algorithms such as live variable analysis and manually formulate rules to infer variable tpyes. TypeArmor [9] and τCFI [10] use live variable analysis and heuristic methods to recover function signatures. Lin and Gao [34] investigates the effect of optimization level on function signature recovery.

Some works introduce machine learning to the task and get some good results. BITY [35] uses the support vector machine (SVM) to classify type inference. TYPEMINER [36] uses both the Random Forest classifier and linear SVM to recover the type in multiple steps. CATI [37] uses CNNs, combined with assembly context as assistance to locate and infer variable types.

B. Input for binary analysis

Choosing an appropriate input is an important step for the success of binary analysis. However, due to the complexity and variety of binary analysis tasks, different inputs are used for different models and tasks to achieve the best results.

MECS [38] directly detect malicious executables with byte sequences. Rosenblum et al. [39] incorporates both idiom features and control flow structure features to identify function entry points. SMIT [40] extracts the function-call graph from a binary program and uses the graph matching algorithm to determine program similarity. ORIGIN [41] extracts significant features from binary programs and recovers provenance with the conditional random field. MutantX-S [42] extracts representative features from malware samples to cluster malware into families.TEDEM [43] automatically finds bugs with bug signatures. Pewny et al. [44] lifts binary codes to the intermediate representation (IR) to obtain the semantics at a basic block level. Genius [45] converts the control-flow graphs into vectors and achieves realtime bug search. RENN [46] learns memory alias dependencies with the binary encoding of instructions and memory region information. XDA [47] learns different contextual dependencies and disassembles with raw machine code. ASTERIA [48] uses the abstract syntax tree (AST) to detect similarity with Tree-LSTM. jTrans [49] tokenizes raw assembly codes and embeds control flow information to detect binary code similarity.

C. Multi-task learning

Multi-task learning is a common effective machine learning architecture where multiple tasks are solved simultaneously. Due to domain information sharing between different tasks, the models are more generalized and robust compared with single-task learning [50].

Many researchers try to improve the MTL performance by modifying the architecture. Misra et al. [51] proposes "cross-stitch", a new sharing unit, to learn a combination of shared and task-specific representations. MRN [52] alleviates negative-transfer and under-transfer by jointly learning features and task relationships. Deep-AMTFL [53] prevents negative transfer by introducing an asymmetric autoencoder term. SNR [54] modularizes the shared low-level hidden layers into multiple layers and controls the connection of subnetworks to improve accuracy and maintain efficiency.

MTL shows excellent results in many applications. Giri et al. [55] uses MTL for speech recognition in reverberant environments. Isonuma et al. [56] addresses document summarization in the framework of MTL. Zou et al. [57] utilize MTL for web searching. Zhou et al. [58] improves the robustness of machine translation with MTL. MKM-SR [59] use MTL for the session-based recommendation. Wang et al. [60] proposed an MTL approach for code understanding. Xie et al. [61] design an MTL method for code summarization. MTLFace [62] alleviates the effect of age variation in face recognition with MTL. DeepCVA [63] uses the MTL model to assess software vulnerabilities with better performance and less time.

VIII. CONCLUSION

In this paper, we present an MTL-GRU model with selective inputs to accomplish function signature recovery and reduce resource consumption. Based on the intuition of information bias, we used two selection strategies that are input length selection and input position selection. Experimental results verify our intuition that most information about function signature is gathered at the head of the function. In addition, motivated by the relationship between the function signature recovery tasks, we make the best use of the correlated information with multi-task learning. As a result of the selective inputs and multi-task learning, our model improves recovery

accuracy and greatly reduces resource consumption both in time and storage size.

ACKNOWLEDGMENT

We sincerely thank the anonymous reviewers for their valuable comments helping us to improve this work. We are grateful to Zhongling He for his contributions and suggestions to the system construction and algorithm design in the early stage of the project. This work was supported in part by grants from the Chinese National Natural Science Foundation (61272078, 62032010, 62172201), the program B for Outstanding Ph.D. candidate of Nanjing University.

REFERENCES

[1] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International conference on information systems security*, pp. 1–25, Springer, 2008.

[2] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 225–236, 2009.

[3] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 117–128, 2009.

[4] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 63–72, 2011.

[5] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 299–308, 2012.

[6] M. Zhang and R. Sekar, "Control flow integrity for {COTS} binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 337–352, 2013.

[7] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy*, pp. 559–573, IEEE, 2013.

[8] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries.," in *NDSS*, 2015.

[9] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 934–953, IEEE, 2016.

[10] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, "τ cfi: Type-assisted control flow integrity for x86-64 binaries," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 423–444, Springer, 2018.

[11] Y. Lin, X. Cheng, and D. Gao, "Control-flow carrying code," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pp. 3–14, 2019.

[12] G. Balakrishnan and T. Reps, "Divine: Discovering variables in executables," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 1–28, Springer, 2007.

[13] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," tech. rep., California Univ Berkeley Dept of Electrical Engineering and Computer Science, 2009.

[14] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, pp. 1–1, 2010.

[15] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[16] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pp. 51–60, 2013.

[17] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 99–116, 2017.

[18] Y. Lin, D. Gao, and D. Lo, "Resil: Revivifying function signature inference using deep learning with domain-specific knowledge," in *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy*, pp. 107–118, 2022.

[19] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pp. 103–111, 2014.

[20] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[22] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] F. Chollet *et al.*, "Keras." https://github.com/fchollet/keras, 2015.

[25] "Gnu binutils." https://www.gnu.org/software/binutils/.

[26] "The dwarf debugging standard." https://dwarfstd.org/.

[27] K. Han, D. Yu, and I. Tashev, "Speech emotion recognition using deep neural network and extreme learning machine," in *Interspeech 2014*, 2014.

[28] Q. Jin, C. Li, S. Chen, and H. Wu, "Speech emotion recognition with acoustic and lexical features," in *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pp. 4749–4753, IEEE, 2015.

[29] M. S. Akhtar, D. S. Chauhan, D. Ghosal, S. Poria, A. Ekbal, and P. Bhattacharyya, "Multi-task learning for multi-modal emotion recognition and sentiment analysis," *arXiv preprint arXiv:1905.05812*, 2019.

[30] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas, "An investigation of byte n-gram features for malware classification," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 1–20, 2018.

[31] M. Mohtarami, R. Baly, J. Glass, P. Nakov, L. Màrquez, and A. Moschitti, "Automatic stance detection using end-to-end memory networks," *arXiv preprint arXiv:1804.07581*, 2018.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR (Poster)*, 2015.

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[34] Y. Lin and D. Gao, "When function signature recovery meets compiler optimization," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 36–52, IEEE, 2021.

[35] Z. Xu, C. Wen, and S. Qin, "Learning types for binaries," in *International Conference on Formal Engineering Methods*, pp. 430–446, Springer, 2017.

[36] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "Typeminer: Recovering types in binary programs using machine learning," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 288–308, Springer, 2019.

[37] L. Chen, Z. He, and B. Mao, "Cati: Context-assisted type inference from stripped binaries," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 88–98, IEEE, 2020.

[38] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, (New York, NY, USA), p. 470–478, Association for Computing Machinery, 2004.

[39] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Learning to analyze binary computer code," in *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, p. 798–804, AAAI Press, 2008.

[40] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference*

on Computer and Communications Security, CCS '09, (New York, NY, USA), p. 611–620, Association for Computing Machinery, 2009.

[41] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), p. 100–110, Association for Computing Machinery, 2011.

[42] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "MutantX-S: Scalable malware clustering based on static features," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, (San Jose, CA), pp. 187–198, USENIX Association, June 2013.

[43] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, (New York, NY, USA), p. 406–415, Association for Computing Machinery, 2014.

[44] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, pp. 709–724, 2015.

[45] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), p. 480–491, Association for Computing Machinery, 2016.

[46] D. Mu, W. Guo, A. Cuevas, Y. Chen, J. Gai, X. Xing, B. Mao, and C. Song, "Renn: Efficient reverse execution with neural-network-assisted alias analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 924–935, 2019.

[47] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021.

[48] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 224–236, 2021.

[49] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "Jtrans: Jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, (New York, NY, USA), p. 1–13, Association for Computing Machinery, 2022.

[50] C. Mao, A. Gupta, V. Nitin, B. Ray, S. Song, J. Yang, and C. Vondrick, "Multitask learning strengthens adversarial robustness," in *Computer Vision – ECCV 2020* (A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, eds.), (Cham), pp. 158–174, Springer International Publishing, 2020.

[51] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert, "Cross-stitch networks for multi-task learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[52] M. Long, Z. CAO, J. Wang, and P. S. Yu, "Learning multiple tasks with multilinear relationship networks," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[53] H. B. Lee, E. Yang, and S. J. Hwang, "Deep asymmetric multi-task feature learning," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 2956–2964, PMLR, 10–15 Jul 2018.

[54] J. Ma, Z. Zhao, J. Chen, A. Li, L. Hong, and E. H. Chi, "Snr: Sub-network routing for flexible parameter sharing in multi-task learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 216–223, Jul. 2019.

[55] R. Giri, M. L. Seltzer, J. Droppo, and D. Yu, "Improving speech recognition in reverberation using a room-aware deep neural network and multi-task learning," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5014–5018, 2015.

[56] M. Isonuma, T. Fujino, J. Mori, Y. Matsuo, and I. Sakata, "Extractive summarization using multi-task learning with document classification," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, (Copenhagen, Denmark), pp. 2101–2110, Association for Computational Linguistics, Sept. 2017.

[57] B. Zou, V. Lampos, and I. Cox, "Multi-task learning improves disease models from web search," in *Proceedings of the 2018 World Wide Web Conference*, WWW '18, (Republic and Canton of Geneva, CHE), p. 87–96, International World Wide Web Conferences Steering Committee, 2018.

[58] S. Zhou, X. Zeng, Y. Zhou, A. Anastasopoulos, and G. Neubig, "Improving robustness of neural machine translation with multi-task learning," in *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, (Florence, Italy), pp. 565–571, Association for Computational Linguistics, Aug. 2019.

[59] W. Meng, D. Yang, and Y. Xiao, "Incorporating user micro-behaviors and item knowledge into multi-task learning for session-based recommendation," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '20, (New York, NY, USA), p. 1091–1100, Association for Computing Machinery, 2020.

[60] D. Wang, Y. Yu, S. Li, W. Dong, J. Wang, and L. Qing, "Mulcode: A multi-task learning approach for source code understanding," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 48–59, 2021.

[61] R. Xie, W. Ye, J. Sun, and S. Zhang, "Exploiting method names to improve code summarization: A deliberation multi-task learning approach," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 138–148, 2021.

[62] Z. Huang, J. Zhang, and H. Shan, "When age-invariant face recognition meets face age synthesis: A multi-task learning framework," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7282–7291, June 2021.

[63] T. H. Minh Le, D. Hin, R. Croft, and M. Ali Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 717–729, 2021.