Anonymous *Cybersecurity* (2021) 4:5
https://doi.org/10.1186/s42400-021-00070-0

Cybersecurity

RESEARCH Open Access

Neutron: an attention-based neural decompiler

Ruigang Liang[1,2*], Ying Cao[1,2], Peiwei Hu[1,2] and Kai Chen[1,2*]

Abstract

Decompilation aims to analyze and transform low-level program language (PL) codes such as binary code or assembly code to obtain an equivalent high-level PL. Decompilation plays a vital role in the cyberspace security fields such as software vulnerability discovery and analysis, malicious code detection and analysis, and software engineering fields such as source code analysis, optimization, and cross-language cross-operating system migration. Unfortunately, the existing decompilers mainly rely on experts to write rules, which leads to bottlenecks such as low scalability, development difficulties, and long cycles. The generated high-level PL codes often violate the code writing specifications. Further, their readability is still relatively low. The problems mentioned above hinder the efficiency of advanced applications (e.g., vulnerability discovery) based on decompiled high-level PL codes.

In this paper, we propose a decompilation approach based on the attention-based neural machine translation (NMT) mechanism, which converts low-level PL into high-level PL while acquiring legibility and keeping functionally similar. To compensate for the information asymmetry between the low-level and high-level PL, a translation method based on basic operations of low-level PL is designed. This method improves the generalization of the NMT model and captures the translation rules between PLs more accurately and efficiently. Besides, we implement a neural decompilation framework called Neutron. The evaluation of two practical applications shows that Neutron's average program accuracy is 96.96%, which is better than the traditional NMT model.

Keywords: Decompilation, LSTM, Attention, Translation

Introduction

Decompilation aims to convert compiled low-level PL, such as executable programs or assembly code, in intermediate representation into functionally equivalent high-level PL, which is friendly to read. Decompilation facilitates the tedious task of manual malware reverse engineering, allowing the use of source-code-based security tools on binary code, such as tools to find vulnerabilities, perform taint tracking. Unfortunately, conventional decompilation tools mainly rely on structured analysis methods such as pattern matching, inserting new rules, or decompiling new PL that requires high costs. Furthermore, existing decompilers usually generate codes that do not conform to standard idioms or cannot be parsed, so that there are many problems in manual or automated analysis. Machine translation principles based on deep neural networks (DNN) automatically learn and extract related programs from code data. It breaks through the bottleneck of decompilation technology that relies heavily on experts to write rules and thus is time-consuming. The NMT-based malicious code detection (Peng et al. 2014; Yadegari et al. 2015; Yakdan et al. 2016), analysis and patching (Yakdan et al. 2016) vulnerability discovery (Li et al. 2018; Heo et al. 2017) and exploit (Wang et al. 2018; You et al. 2017; Zong et al. 2020) have sprung up and have been implemented in engineering applications, providing breakthroughs in cyberspace security technology. Recent work has shown that neural networks are also useful in summarizing source code (Loyola et al. 2017; Allamanis

*Correspondence: liangruigang@iie.ac.cn; chenkai@iie.ac.cn
[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

et al. 2015). The above works indicate that deep learning technologies such as NMT have a vast application range in program analysis.

Several NMT-based approaches to neural decompilation for PL have been proposed to overcome the current bottlenecks faced by rule-based approaches. In these works, decompilation for low-level PL is converted into translation problems between PL, and then decompilation tools are built using NMT technology to enable neural translation from low-level PL to high-level PL. The existing neural decompilation technology has been improved, but they can only achieve accurate semantic recovery for simple functions (e.g., arithmetic operations), but still powerless for complex functions or real-world PL code.

Our approach. We propose a neural program decompilation framework, named Neutron[1]. Neutron mainly consists of three core phrases: Code Preprocessing, Neural Translation, and Function Reconstruction.

In the first phrase, Code Preprocessing is committed to the standardization of PL and helps the model learn the conversion rules between the high-level PL and the low-level PL well. To reduce the decompilation difficulty of the NMT model, we disassemble the binary code into assembly language, which contains richer semantic as well as structural information, and thus is utilized as the target low-level PL. Further, to avoid the adverse effects of identifiers (e.g., variable names), we propose a method to standardize the PL code in model training. Using the regularized low-level PL code and high-level PL code pairs as training sets can effectively reduce the difficulty of model learning conversion rules.

In the second phrase, Neural Translation aims to design a neural decompilation model, which can translate a low-level target PL into a functionally similar C-like high-level PL code. After preprocessing the PL dataset that is generated by cfile (2020) in the first phrase, Neutron trains a neural-based decompilation model AsmTran, which is based on LSTM-Seq2Seq-attention (Luong et al. 2015) architecture. Then we use AsmTran to translate the low-level target PL into a high-level PL while keeping functionally similar.

In the third phrase, Function Reconstruction focuses on restoring the function's dependencies through specific rules. Since AsmTran takes the basic operation of PL as the translation unit, its output is also independent and regularized high-level PL fragments, missing the dependency between variables and sentences in the function. To reconstruct the function's structure, we manually define rules to gradually reconstruct the complete function structure from data flow recovery, control flow recovery, as well as parameters and return value recovery.

[1]Neutron (Neural translator for binary code)

We implement Neutron on the base of the attention-based NMT mechanism in the tensor2tensor framework (tensor2tensor 2020) and evaluate the performance using real-world applications. The results show that Neutron achieves an average accuracy of 96.96% on three real-word projects and three different tasks. The results demonstrate that the output of Neutron recovers functionality and improve readability significantly.

Contributions. The contributions of this paper are outlined as follows:

- *New technique.* We implement our technique in a framework called Neutron. Neutron has general applicability and higher readability in benchmarks of various levels of complexity and real-world projects, which also provides superior performance compared to existing neural decompilation tools. Neutron overcomes several vital challenges that prior research has not effectively solved, including (i). designing a neural decompilation mechanism based on PL basic operations, (ii). introducing an iterative error correction method to improve the accuracy of the model, and (iii). using a rule-assisted technique to recover the function structure, such as control flow and data flow of the function. Neutron can be easily ported to other types of high-level PL decompilation tasks with negligible engineering overhead.
- *New Understanding.* Our study suggests that it is feasible to apply the NMT model for natural language to the PL decompilation task. We design a new neural translation mechanism based on the basic operations of PL, which is more conducive to the model's learning of decompilation rules. The mechanism can make the NMT model directly competent for the decompilation task of PL code and effectively improves the generalization ability of the Neutron.

Road Map. The rest of the paper is organized as follows: Background and related work section presents the background and prior work related to our research. Overview section describes the summarize of our research. Design and implementation section elaborate the design and implementation. Evaluation section reports our experimental results. Discussion section discusses the limitations of our approach and potential future research, and Conclusion section concludes the paper.

Background and related work

Conventional Decompilation

Conventional decompilation mainly depends on computer scientists to define decompilation rules through control flow analysis, to realize the conversion of a low-level PL into intermediate language or high-level language representation that is more convenient for humans read (Ďurfina et al. 2011; Ďurfina et al. 2013; Yakdan et al. 2016; Yakdan et al. 2015; Brumley et al. 2013). Ďurfina et al.

(2011) outlined the development history of decompilation technology for more than 50 years.

The current representative decompilers mainly include Phoenix (Brumley et al. 2013; Hex-Rays 2020), RetDec (Křoustek et al. 2017), and Ghidra (2020). Both Hex-Rays and Phoenix rely on pattern matching to identify the program's advanced control flow structure and change the control flow graph (CFG) of the program. Hex-Rays can display the C-like code generated by decompilation in the window, and jump to the function body window by clicking the function name. The segments match patterns are known to originate from specific control flow structures. When faced with non-trivial code, decompilation often fails, and a large number of `goto` statements are used to simulate the control flow of low-level PL. Although it is semantically equivalent to the original low-level PL code, it is difficult to read and relatively inefficient. In response to this problem, scientists have targeted goto-free for research, such as DREAM++ (Yakdan et al. 2016; Yakdan et al. 2015), which can restore all control structures in binary programs and generate structured decompiled codes without any `goto` statements.

RetDec (Křoustek et al. 2017) is a redirectable machine-code decompiler based on LLVM and developed by the Czech security company Avast in 2017. It aims to become the first "universal" decompiler that is capable of supporting multiple architectures and multiple languages. However, according to data released by Avast, the development of RetDec requires a team of 24 developers to develop for seven years to complete (Avast Retargetable Decompiler IDA Plugin 2020). Ghidra (2020) is a software reverse engineering (SRE) framework developed by the National Security Agency (NSA) for the NSA's network security tasks. It is used to assist in analyzing malicious code, viruses and other malicious software, and understanding its network and system *Potential loopholes*. Ghidra contains hundreds of functions (e.g., disassembly and decompilation) and supports multiple processor instruction sets and executable formats.

Because traditional decompiler is based on hand-made rules designed by experts, and these rules are difficult to develop and error-prone. Usually only part of the known control flow structure can be captured, which lead to poor scalability as well as the slow and costly development of decompilers. Under the background of the successful development of deep learning, especially NMT technology, which brings new ideas to decompilation technology to break through the current bottleneck. Using the powerful learning and expression capabilities of deep learning models to automate the decompilation process can be significant, improving the development cycle of decompilation tools, saving R&D costs, and enhance their scalability.

Neural Decompilation

Due to the limitations of the traditional decompilation technology based on rule matching mode, artificial intelligence (AI) technology is used to build an intelligent decompilation mechanism to break through the distinction between code and data, indirect jump and indirect call instructions, self-modifying code recognition, and data type recovery. The bottleneck has now become a research direction in the field of decompilation. The related technology of NMT can be used for code decompilation because assembly program, intermediate code, or high-level PL can also be regarded as a language. Therefore, the problem of decompilation between natural languages can be regarded as translation problems between PL. There has been some work to build neural decompilation tools based on NMT technology (Katz et al. 2018; Katz et al. 2019; Fu et al. 2019).

Levy and Wolf (2017) proposed a method that uses neural networks to predict the alignment between source code and compiled target code, aiming to solve the difference between decompiled and manually written code. The model learns the attributes and patterns in the source code and uses them to generate decompiled output. This work can be extended and applied to situations that are not targeted at traditional decompilers, such as optimizing the readability of decompilation, restoring control flow structure, or variables.

Katz et al. (2018) proposed to use recurrent neural network (RNN) (Pearlmutter 1995) to build a decompiler. They trained an RNN model to convert binary code into C-like code directly and improved syntax and semantic accuracy through post-processing. Unfortunately, their work did not make up for the difference between natural language and PL, resulting in poor decompilation output, and the post-processing method was too simple to guarantee syntax correctness. Recently, (Katz et al. 2019) used Long Short Term Memory (LSTM) (Hochreiter and Schmidhuber 1997) networks to build a decompiler named TraFix. They proposed a way to preprocess assembly language (input) and post-process C language (output), which narrowed the difference between the PL and natural language. However, TraFix performs poorly on decompilation of conditional branches and `loop` statements.

Overview

The overview of Neutron is illustrated in Fig. 1, including three main components: Code Preprocessor, Neural Translator, and Function Reconstructor. Figure 2 shows an concrete example of Neutron's decompilation of low-level PL.

In the Code Preprocessor phrase, Neutron is committed to ASM generation (the step ① in the figure) and Code Standardization. The ASM generation module is

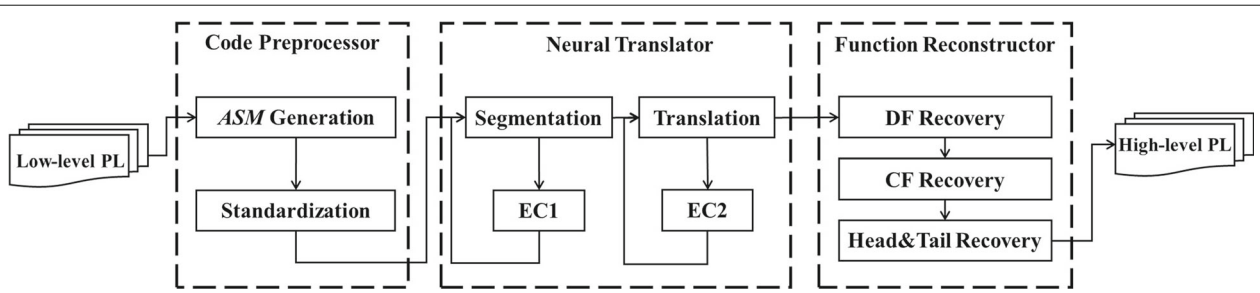

Fig. 1 An overview of Neutron

mainly responsible for disassembling the binary code into assembly code. Because assembly code has richer semantic and structural information than binary code, and the disassembly technology is very mature, using assembly language as a low-level PL can effectively reduce the difficulty of NMT model learning decompilation rules. The standardization module focuses on the regular processing of identifiers, numbers, etc. in the PL code, which reduces its impact on model training and translation to learn better the conversion rules between low-level PL and

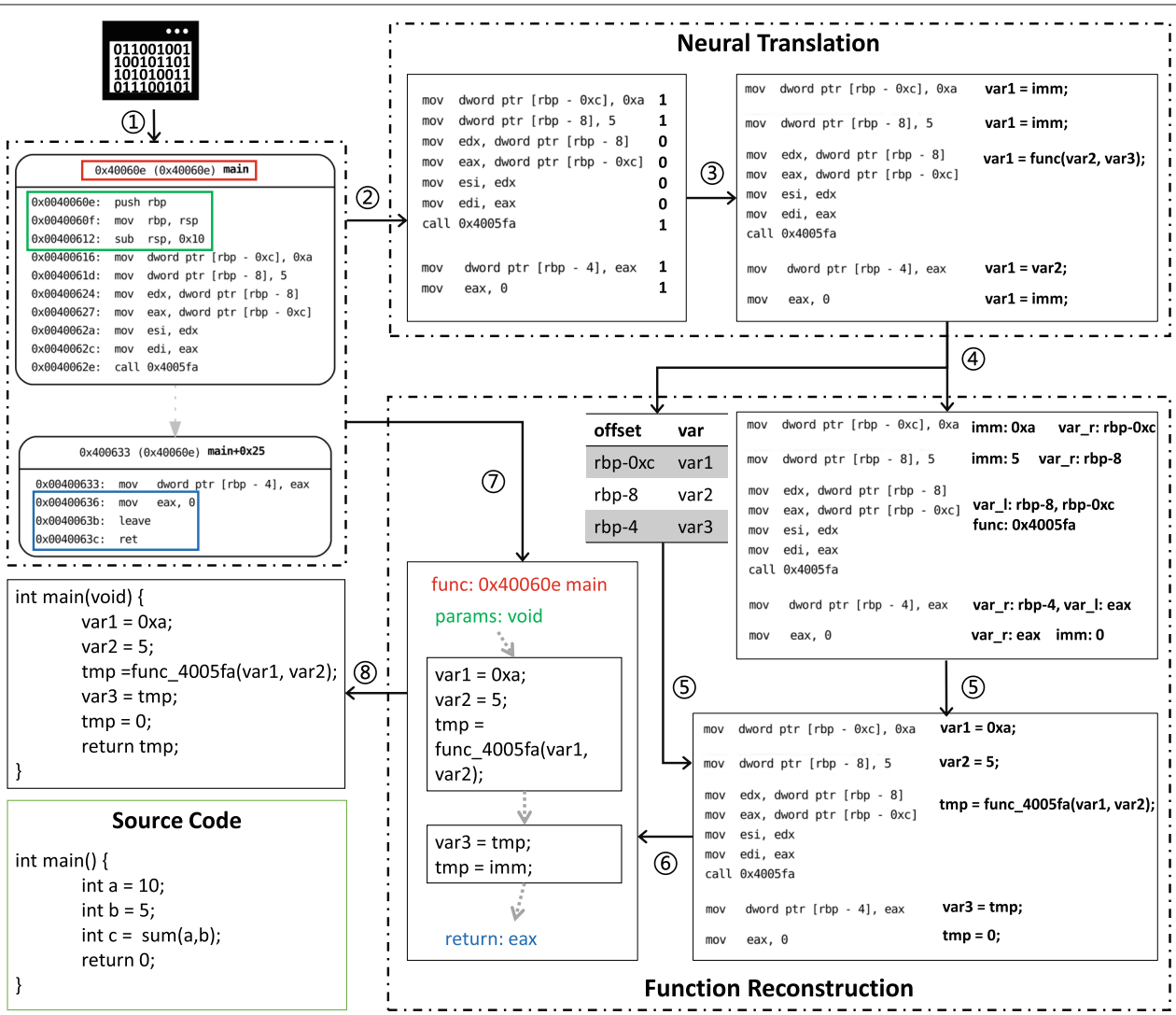

Fig. 2 An example of low-level PL decompilation by Neutron

high-level PL. Since the standardized operation description is relatively clear and simple, we did not reflect this part in the Fig. 2. Besides, in the training phase of the AsmTran model, to improve the complexity and robustness of the training set and ensure the balance of various data types in the training set, we design a data augmentation method to expand the data set to four to five times, so that the model can better learn the translation rules between PL pairs.

In the Neural Translator phrase, we design a new neural network architecture AsmTran that is suitable for learning conversion rules between low-level PL and high-level PL, which trains based on the regularized dataset to make it accurately learn the conversion rules. The AsmTran model is mainly divided into two sub-models. The first sub-model is a text classification model (step ②) aiming to perform fine-grained code segmentation for low-level PL based on basic operations. The second sub-model is an NMT model (step ③), which takes each basic operation of the target low-level PL as input and outputs its corresponding high-level PL. The translation unit based on basic operations can reduce the difficulty of learning decompilation rules for the AsmTran model, making the NMT model suitable for PL decompilation tasks and improving the model's generalization ability. It is worth mentioning that we introduce the iterative error correction (EC) mechanism in both sub-models in the training phrase. The prediction errors in the two sub-models' output are fed back to the sub-model itself to improve the AsmTran's performance through the manual definition of judgment and EC rules. Finally, Neutron utilizes AsmTran to decompile the target low-level PL code and outputs a C-like high-level PL code.

Since AsmTran translates the basic operation of the code as a unit, and the low-level target PL undergoes regularization processing, its output result is also a regularized high-level PL fragment corresponding to the function is no actual function structure. Therefore, we design a function reconstruction method, which mainly realizes the restoration of data flow recovery (step ④ and ⑤), control flow recovery (step ⑥), parameters and return value Recovery (step ⑦) by manually defining rules, and further completes the reconstruction of the function structure (step ⑧), which effectively improves the readability of the decompiled high-level PL.

Design and implementation

Code preprocessing

As mentioned above, the main challenge of applying the NMT model in the decompilation technology is to preprocess the PL to meet the training data input requirements of the NMT model without losing its information. Unlike natural languages, high-level PL is hierarchical, such as C, C++, etc. It is not very practical to apply the serialized NMT model directly to PL's decompilation because it is challenging to learn the syntax structure of high-level PL. To overcome this challenge, we propose a forward-looking code preprocessing method. First, we introduce a disassembly mechanism to disassemble the binary code and use assembly language as a low-level PL for decompilation. The reason is that the binary code contains less semantic information. From the direct translation of the binary language to the high-level PL like C code, it is difficult for the NMT model to learn the conversion rules. Secondly, in order to improve the training efficiency of the model and the accuracy of the translation, we regularize the code, which learned from the word segmentation method in natural language process (NLP). And then realize the word segmentation operation of PL by setting rules to prevent the accuracy of translation from being affected by model segmentation errors. Finally, in order to ensure the balance of various statement types, such as arithmetic operations, logical operations, comparison operations and function calls in the training set. We design a data augmentation technique, and at the same time increase the scale and robustness of the training set. This ensures that the model can better learn the translation rules between low-level PL and the high-level PL.

ASM generation

Compared to binary code, assembly code has richer semantic and structural information, and the existing disassembly tools, such as objdump, angr (Shoshitaishvili et al. 2016), etc., can better convert the binary code into a functionally corresponding assembly program. Angr is an open-source python framework for analyzing binary files, which contains a flexible interface applicable to various analysis tasks. We use the disassembly and CFG analysis module `CFGEmulated` in angr to perform fundamental analysis on the binary code and obtain the assembly code segments and CFG of all functions.

Standardization

To enable the NMT model to learn better the conversion rules between low-level PL and high-level PL, we standardize the PL code pairs in the training set.

```
1 mov  edx, DWORD PTR -4[rbp]
2 mov  eax, DWORD PTR -8[rbp]
3 add  eax, edx
4 mov  DWORD PTR -12[rbp], eax //var3 = var1 + var2;
```

Listing 1 An example of low-level PL in Intel format

Low-level PL Standardization. Listing 1 is an example of assembly code in Intel format[2] with the expression $var_2 = var_0 + var_1$. As assembly code has fixed and fewer syntax rules, we define the rules manually for word seg-

[2]The current mainstream assembly code has two formats, Intel and AT&T, which differ only in syntax. We use the former in this paper.

mentation, separating instruction mnemonics, numbers, and symbols with `space`. We also use the symbol '`;`' to separate each instruction. For example, the instruction form of line 2 in Listing 1 after regularization processing is `mov eax , DWORD PTR -8 [ rbp ] ;`.

High-level PL Standardization. We choose the C program as the target PL for decompilation. The main reason is that C high-level PL has higher readability, and compared with object-oriented PL, its syntax is more straightforward and comfortable for the NMT model to learn. However, word segmentation is still required due to the C-like high-level PL's own syntax rules and special symbols. We use `space` to separate reserved words, variable names, and symbols. The C-like high-level PL contains many user-defined elements such as variable names, strings, constants, etc. These elements will cause the explosion of the model word list on the one hand, and also affect the convergence of the model on the other hand. Therefore, we also standardize these elements: First, we rename the variables in each sample from var_0 to var_n; Secondly, since the generative model in NLP cannot handle the replication problem, we use the mark symbol `imm` to replace elements such as strings and constants.

```
1 mov  edx, DWORD PTR -4[rbp]
2 mov  eax, DWORD PTR -8[rbp]
3 add  edx, eax                 //tmp = var1 + var2;
4 mov  eax, DWORD PTR -12[rbp]
5 imul eax, edx                 //tmp = tmp * var3;
6 mov  DWORD PTR -16[rbp], eax  //var=tmp;
```

Listing 2 An example of ternary operation of low-level PL

Training data augmentation

The complex statements in the high-level PL can be split into a combination of multiple basic operations. Therefore, we use the random code generation tool (cfile 2020) to randomly generate many basic operation statements, such as Listing 1, to obtain low-level PL and high-level PL pairs, then mix various types of sentences in proportion as the raw data set to ensure the balance of the data set. However, program statements are usually like Listing 2 in reality. In real-world PL, the variables temporarily stored in the register can be used in later operations to reduce the memory read and write operations. Therefore, the actual segmentation's code fragments may not contain memory-related instructions. Model trained with raw data can not handle this problem. In order to solve this problem, we use the method of deleting memory operation instructions to process the raw data set and obtain an expanded data set after data augmentation (the sample size can reach four to five times the raw data set), make it meet the ability to process a small fragment after compilation. Also, the data set's existing sample data may not cover all the program forms in the real world, so there may be deviations in the offset addresses of registers or variables. In order to increase the robustness of the model and enable it to better handle situations that did not appear in the training set, we perform random masking operations on the words in the raw data set and the extended data set by covering 20% of the words in some sentences, and get the mask data set. Figure 3 shows an example of data augmentation. In the figure, augmentation 1 and 2 are an augmented example of deleting memory from the raw data. While augmentation 3 and 4 are examples of augmentation to block certain words or instructions. Finally, we construct the training set by organically fusing the above three data sets. Benefit from the training data augmentation phrase, the average accuracy of our Neutron increased by 73.02% during evaluate different tasks.

Neural Translation

After preprocessing the PL dataset with the preprocessor, Neutron utilizes the regularized dataset as the training set for our neural decompilation model, which is based on the idea of attention-based NMT mechanism. The detailed design is as below.

Segmentation

Similar to natural language translation, the decompilation of low-level PL to high-level PL can also be seen as a translation problem between two natural languages. However, because PL has stricter syntax rules and information asymmetry between PLs, it is more difficult to translate between PLs than natural languages. Besides, the issue of the length of high-level PL statements also needs consideration.

Because combinations between expressions in PL are even much more diverse than in natural language, the data set cannot include all possible combinations in the code. Hence, it poses a more significant challenge to the translation model's generalization ability. Fortunately, since PL has strict syntax rules and the number of rules is relatively small, we modify and optimize the NMT model in combination with the rules to make it be able to meet the decompilation task of PL. We carry out a more fine-grained division of a code line, reducing the translation unit of the NMT model from a code sentence line to basic operation. A line of code is usually composed of one or more phrases. These phrases are the basic types of operations in the code, such as `unary` operations, binary operations, function calls, etc. Taking the basic operation of PL as the translation unit of the NMT model can effectively improve the model's generalization while reducing model learning difficulty.

The fine-grained code segmentation can be regarded as a text classification task. We use the sequence model LSTM encoder (tensor2tensor 2020) to classify each instruction. According to the model's tags output, low-level PL code fragments can be converted into a combi-

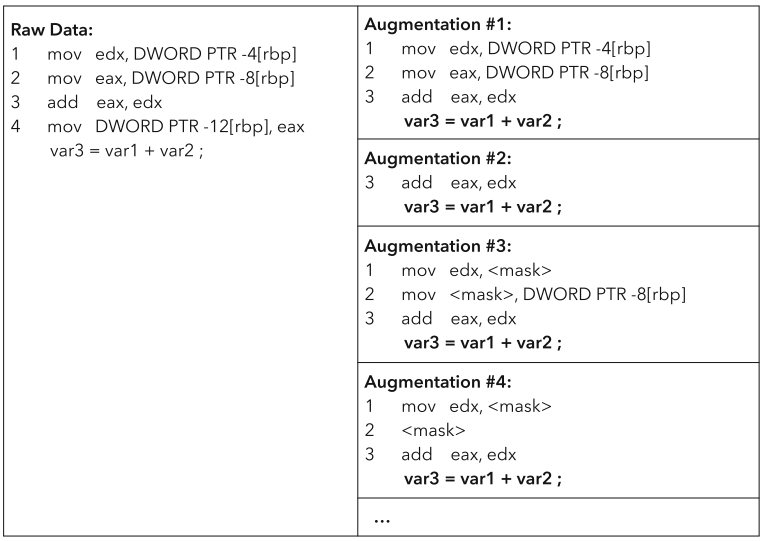

Fig. 3 An example of training data augmentation

nation of several basic operations. Listing 2 is the code fragment of the expression $var_4 = (var_1 + var_2) * var_3$. After fine-grained segmentation, the output label is seen as `<0,0,1,0,1,1>`, which is obviously divided into three basic operations. Each basic operation would be used as the input of the translation model.

Translation

After fine-grained segmentation of low-level PL, we divide a line of code into code fragments with considering basic operations as units. The code fragments are similarly treated as units for subsequent decompilation. In this way, the difficulty for the translation model to learn low-level PL and high-level PL conversion rules is significantly reduced, and it can cope with PL structures that do not appear in the training set, while effectively improving the generalization ability of the AsmTran model.

The decompilation task of PL is similar to machine translation in NLP, which is a text generation task of sequence-to-sequence (Seq2Seq) (Sutskever et al. 2014). The Seq2Seq model is a particular type of RNN architecture, usually used to solve complex language problems such as machine translation (Sutskever et al. 2014; Wu et al. 2016), text summarization (Shi et al. 2018), and question answering (Yang et al. 2016). The most common Seq2Seq model architecture is encoder-decoder architecture. The encoder converts the input sequence into a fixed-length vector encoding, while the decoder decodes the fixed vector and converts it into an output sequence, where the encoder and decoder are mostly LSTM models. In the Seq2Seq model, since the encoder transforms the variable-length input sequence into a fixed-length semantic vector, there is a loss of information in the encoding process, and the longer the sentence, the more apparent. Also, in the decoder operation, the output at each moment uses the same context vector in the decoding process, so there is a specific deviation in the prediction result.

In order to solve the above problems, researchers introduced the attention mechanism into the Seq2Seq model so that the context used by the model when predicting the output at each moment is the context related to the current output. In another word, the weight of the semantic vector changes dynamically according to the predicted vocabulary. The attention mechanism allows the model to assign higher weights to specific parts of the input sequence when decoding instead of focusing only on the last hidden layer's results in the LSTM model. The attention mechanism solves the problem that long-distance information will be weakened in the RNN models and quickly grasp critical points in long texts without losing important information. We introduce the attention-based NMT model (Luong et al. 2015) as the decompilation model, whose architecture is shown in the Fig. 4.

Given an input code $X = (x_1, \cdots, x_m)$, we use $x_i \in \mathbb{R}^d$ to represent the i-th word in the input. The output high-level PL code sequence is defined as $Y : (y_1, \cdots, y_m)$. For the input sequence X, the encoder first maps each word x_i in X to a vector w_i to obtain the model input $W : (w_1, \cdots, w_n)$. The vector e output by the encoder can represent the input low-level PL code sequence's context information. The decoder decodes according to e and the current output sequence and inserts a start tag `<s>` for each source code. The decoder stops decoding when it reaches the terminator `';'`. The initial hidden layer state h_0 of the decoder is calculated according to e. The decoder decodes the hidden state h_i of the i-th word to calculate the probability distribution p_i of the i-th word. Its input is the vector w_{i-1} corresponding to the word $i-1$ and

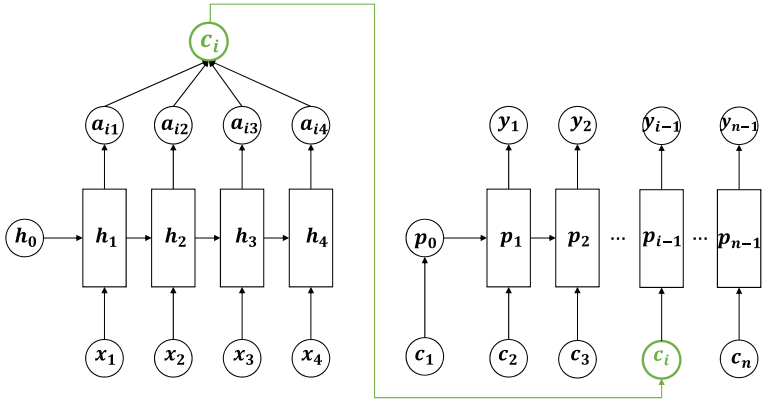

Fig. 4 Attention-based LSTM architecture

the current hidden layer state h_{i-1}, as well as the attention vector c_i.

$$h_0 = LSTM(e, w < s >) \tag{1}$$

$$c_i = \sum_{i=0}^{n} softmax(f(h_{i-1}, e_i))e_i \tag{2}$$

where f is the function of calculating the attention vector.

$$h_i = LSTM(h_{i-1}, [\, w_{x_i-1,c_i}]) \tag{3}$$

$$p_i = softmax(generator(h_i)) \tag{4}$$

where the function $generator : \mathbb{R}^d \Rightarrow \mathbb{R}^v$ (v is the length of the word list), the *softmax* function maps the *generator* input to the (0, 1) interval to obtain the probability distribution of the ith word on the word list. Finally, the word with the highest probability is selected as the decoded word y_i, and the current word can be obtained by mapping $\hat{y}_i$ to the word list.

$$\hat{y}_i = argmax(p_i) \tag{5}$$

In the test phase, *AsmTran* adopts the beam search decoding method (Reddy 1977). Each neuron selects the k outputs with the highest current output probability and transmits them to the next neuron.

AsmTran takes the basic operations of low-level PL (assembly code) as inputs and the high-level PL (C-like source code) as output. We use a mixture of different types of codes as the training set and adjust the proportion of various types of codes according to the translation results to improve the model's accuracy. The size of the training set is 1.53 million. Benefit from the design of *AsmTran*, the accuracy of our approach is increased by 36.11%, compared to the state-of-the-art natural language translator (Luong et al. 2015).

Iterative error correction

Error Correction for Segmentation. In the AsmTran model, the segmentation sub-model may produce some false-positives and false-negatives. EC1 is mainly divided into two parts. First, we use rules to correct some obvious classification errors, including memory write instructions, function call instructions, separate arithmetic, and logic operation are obvious boundaries, while memory read instructions are not boundary instructions. Iterative training enhances the sub-model's classification capabilities by adding the incorrectly classified instructions and their correct labels to the training set.

The other most crucial error is that to speed up code execution, the division operation in some cases is changed into a combination of multiple other basic instructions like `sub`, `add`, `imul`, `shl`, `shr`. These operations may be mistaken for boundary instructions by the segmentation sub-model. Due to divisible optimization instructions' fixed characteristics, they are usually composed of several addition, subtraction, and shift instructions. Therefore, we identify the suspected divisibility code block by defining rules, correcting each instruction's marking, and only mark its last instruction as a boundary instruction.

Error Correction for Translation. In the AsmTran model, there are two main types of errors in the translation sub-model: syntax errors and semantic errors. The syntax error means that the translated code does not conform to the syntax specification of C code. We use regular expressions to design a syntax checker, named CS[3], for the types of statements in the data set to check for syntax errors. For some syntax errors that have little impact on errors, we use rules to correct them, including errors in brackets and commas between parameters in function call statements, errors in the order of operands and operators in arithmetic and logical expressions, and errors in conditional expression ":" and "?".

[3] CS (Check syntax)

Semantic errors mean that the meaning of the sentence obtained after translation is different from the meaning of the original assembly code. It is difficult to check and correct semantic errors, so we correct a few particular semantic errors, including obvious translation errors in arithmetic and logic operations, translation errors in function calls, and an incorrect number of function parameters. We obtain some obvious guidance information from the assembly statement to fine-tune the decompiled code. In the training process, we use the supervision information to add the wrong sentences in the test set and their correct labels to the iterative training of the training set to improve the performance of AsmTran.

Function Reconstruction

After AsmTran has completed the decompilation of the target low-level PL code, we get regularized high-level PL code fragments corresponding to low-level PL's basic operations. However, in the actual source code (function), there are control dependencies and data dependencies between variables and statements. Therefore, we need to restore the function's dependencies through specific rules, further complement the function's head and tail, and finally build a complete function. Our function reconstruction technique is divided into three parts: data flow recovery, control flow recovery, as well as parameters and return value recovery.

Data flow recovery

In the code preprocessing phrase, in order to reduce the influence of the variable names in the PL on the subsequent model translation, we have carried out regular operations on them, so the variable names of the decompiled high-level PL are all in the form of var_i, which causes many obstacles to the readability and understandability. Therefore, we need to reconstruct the variable name in the function to restore its data dependency.

Our method is mainly composed of three steps: (1) Variable extraction. We extract the operands corresponding to the low-level PL code variables, build a hash table, and assign variable names to each operand, starting from var_0 to var_n. (2) Ingredient identification. For each basic operand of low-level PL, we identify its position in the current code fragment and the bound operation. The basic rules are as follows: ① `mov` read memory operation, marked as (right, '`mov`'); ② `mov` write memory operation, marked as (left, '`mov`'); ③ other instructions read memory operation, marked as (right, opcode); ④ Write memory operation for other instructions, marked as (left, opcode). During this step, we can associate the meaningless variables var_i and `imm` in the decompiled code sketch with the offset address and immediate value in the low-level PL code to form a mapping relationship. For example, the second code snippet `mov dword ptr [rbp - 8], 5` correspond to var_1 `= imm`, so the mapping relationship `imm: 5, var1: rbp-8` can be obtained. (3) Variable name restoration. According to the corresponding relationship between the variable name and the offset address obtained in the first step, we replace the variable's position in the code sketch corresponding to the offset address in the second step with the new variable name. For example, the variable name corresponding to `rbp-8` in the entire function should be var_2 in Fig. 2, so we use var_2 to replace var_1. Similarly, we use 5 to replace `imm`, then we get var_2 `= 5;` after the data flow recovery. For the position where there is no corresponding variable name, this situation is usually caused by the split of a complex sentence, resulting in the lack of memory to read and write instructions. In this case, there is no corresponding sentence in the source code, so we use "`tmp`" as an intermediate variable to replace these positions.

Control flow recovery

In addition to data dependencies, control dependencies exist before code blocks, such as conditional branch structures and loop structures. The control dependent structure recovery constructs the CFG of the recovery function, which is very important. Since we segment the assembly code block of a function, the code recovered by the translation model is only one sentence by sentence, and each sentence is independent of each other, lacking the proper organization structure within the function. In the process of code preprocessing, we use *angr* to obtain the CFG of the function, so we use this CFG to restore the control dependence of the program function after decompilation, which can be divided into two steps: (1) *Basic block internal sequence recovery*. The basic block statements are executed sequentially, so the order between the high-level PL codes can be restored according to the basic block's assembly blocks' sequence. (2) *Jump relationship recovery*. There are a jump relationships between basic blocks. The condition types or the loop types can be judged according to the jump directions. The condition expression is determined according to the last condition expression in the jump block. Loops all use the while loop format. If it cannot be determined, we use the `goto` statement temporarily.

Parameters and return value recovery

Parameters and return values are two critical elements in the code function, and they are the external interface parts of the function. Therefore, accurate identification of parameters and return values helps to analyze the function call relationships of the entire binary code. The return value is usually stored in the `eax` register, located in the last basic block in the assembly code. In the above process, we have translated the last sentence. The final return value helps to analyze the entire binary code's function

call relationship. When the return value is determined, we can judge the entire function's return type based on the type of `eax`. However, it is not easy to find the parameter list directly from the function. We determine the parameter list of the target function through other assembly code fragments that call the function. For example, the GCC X86-64 compiler prefers to use registers, such as `rdi`, `rsi`, and `rdx`, to pass parameters, and then pass them using *program stack* if the number of parameters are more than 7.

Evaluation

Experimental Setup

We evaluate the performance of Neutron on a variety of benchmarks with real-world applications and different tasks, as shown in Table 1. All the experiments are performed on a 64-bit server running Ubuntu 18.04 with 16 cores (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz), 128GB memory, 2TB hard drive and 5 GPUs (2 GTX Titan-V GPU and 3 GTX Titan-X GPU).

Training data generation. To build a dataset for training, and testing of Neutron, we randomly generate 180,000 pairs of high-level PL codes with corresponding assembly codes as the raw data set. The program is compiled using GCC 9.3.0, with all optimization options disabled. Furthermore, we mark the basic operations' boundary instructions in all the assembly code blocks of the raw data set to obtain the segmentation sub-model training data set. For the translation sub-model, we expand the raw data set based on the above data augmentation technique. We obtain 1.53 million pairs of high-level PL codes with corresponding low-level PL codes as the training set.

Benchmarks. We evaluate the performance of Neutron using real-world applications. Particularly, we select three real-world projects (Warren 2012; Karel the Robot 1995; Yang et al. 2011) and three different tasks of different statement types, such as arithmetic operations, logical operations, comparison operations, and function calls, which generated using real libraries (Math c++ library 2020). (1) *Hacker's Delight loop-free programs* (Warren 2012) constructed by Schkufza et al. (2013), which is used to encode complex algorithms as small, loop-free sequences of bit-operated instructions. (2) *Karel.* (Karel the Robot 1995) is a C-based library for control robot actions, which contains more function call statements. (3) *Csmith.* Csmith (Yang et al. 2011) is a tool for testing compilers or other tools that process C code. We randomly generate various types of C programs of different complexity by the Csmith 2.3.0. Liu and Wang (2020) also used Csmith to evaluate existing conventional decompilers. (4) *Math.* We generate code by the `math.h` library (Math c++ library 2020), which contains more function call statements. (5) *Normal Expression (NE).* We use NE to generate code containing arithmetic, logic, and comparison expressions. (6) *Math + NE.* We also utilize the `math.h` library and NE to generate more complex code. Each function contains the above four types of statements. The most recent work Coda (Fu et al. 2019) also exploited these similar methods to evaluate its performance.

Effectiveness

We measure the effectiveness of Neutron. The effectiveness is dependent on whether the target low-level PL code is translated into the high-level PL code with similar functionality.

Performance on real-world projects. We evaluate the performance of Neutron using the five real-world applications, as mentioned above. To ensure the accuracy and objectivity of evaluation results, we remove duplicates of those data. From Table 1, we can find that Neutron is very accurate. Such an accurate model enables Neutron to have high performance. There are lines of C program code composed of many mixed comparison, logical, and arithmetic operations, which directly leads the model unable to translate such overly complex sentences accurately. Compared with using the LSTM-Seq2Seq-attention model (Luong et al. 2015), our approach achieves 36.11% higher accuracy on average, which reflects that the attention-based NMT method fails to learn the conversion rules between PL pairs effectively. Besides, our approach could improve 74.71% on average than using LSTM. This is mainly because LSTM can not handle long-term information. The above result shows that our Neutron has a significant performance.

Performance on different tasks. Based on the real-world code library, we generate code blocks of different data types, such as arithmetic operations, logical operations, comparison operations, and function calls, to evaluate Neutron's translation effects on different data types. Because the PL code is a combination of various data types, the model's translation accuracy of data types can also reflect low-level PL decompilation performance. The results of Neutron's accuracy are shown in Table 1. For Math and NE tasks, the native attention-based LSTM

Table 1 Performance on different benchmarks

	Hacker's Delight	Karel	CSmith	Math	NE	Math+NE
Neutron	100%	100%	95.45%	98%	97.4%	97.00%
LSTM+attention	8.69%	81.05%	73%	80%	71.79%	56.67%
LSTM	0	0	21.7%	37%	31.66%	49.18%

performs well, but it appears weak for the more complex Math+NE tasks. In contrast, Neutron has excellent performance (accuracy rate higher than 97%) for the above three tasks of different complexity. Since we apply the code segmentation method, the accuracy of Neutron is not limited by the length of a single statement of the code. However, the related work Coda (Fu et al. 2019) is greatly affected by the length of the single code sentence. When the code length increases to 30, Coda's accuracy drops by an average of 5.4% - 13.5%.

Runtime Performance

Time cost of training. We calculate the training time for Neutron using the 1.6 million datasets. The time spent by the model for every 100 steps is 16 seconds, and the model completes the training task with a total of 8,000 steps. Therefore, the training time for Neutron is about 0.5 hours.

Time for translation. Regarding the time of translation, we randomly select 100 low-level PL code fragments (corresponding to a line of high-level PL code) from the test set for the translation efficiency evaluation. After 100 random experiments, it shows that the average testing time of *Neutron* is 1.01 seconds.

Influence of Parameters

Impact of training data augmentation

Data augmentation includes expansion and random masking. First, we evaluate the impact of using data augmentation on Neutron's accuracy. Second, we evaluate the effect of different iteration times and mask ratios on our model's accuracy.

Firstly, We evaluate the accuracy of the translation submodel of the AsmTran (without code segmentation) on a data set, consisting of four sentence types: arithmetic operations, logical operations, comparison operations, and function calls. We use the raw data set (R), extended data set (E), and mask data set (M) to train three models and then evaluate the code translation accuracy of these three models. The results are shown in Table 2, which suggests that data augmentation plays an essential role in improving Neutron's accuracy. To separate the effect of EC from the effect of data enhancement, here is the accuracy of the model before EC. The expansion part enhances the ability to deal with incomplete code fragments, and the mask part significantly enhances the model's robustness.

Table 2 Impact of training data augmentation

Training data set	Arithmetic	Logic	Comparison	Function call
M(R)	1.89%	2.35%	0.74%	10.96%
M(R+E)	82.3%	69.8%	60.77%	83.16%
M(R+E+M)	85.2%	72.06%	63.4%	87.41%

Secondly, we evaluate the impact of the number of iterations and mask ratio on the model's accuracy. The results are shown in Fig. 5. When the number of iterations exceeds 5, the accuracy of the model does not improve significantly. The model works better when the number of iterations is 10, and the mask ratio is 0.2. Besides, when there is no iteration, the model's average accuracy is only 1.74%. The more iterations, the larger the number of training sets, and theoretically, the higher the model's accuracy. However, after segmentation, the types of instructions contained in the assembly code block are limited. Too many iterations would cause many repeated data, so we set the upper limit of the number of iterations to 10. Similarly, if the mask ratio is too large, the code sentence's characteristics would be blurred, which is not conducive to the model's convergence. Therefore, we set the upper limit of the mask ratio to 0.2.

Impact of iterative error correction

We use a mixed data set (containing 150,000 pairs of PL codes) constructed by the above four sentence types to evaluate the two EC modules' performance in the AsmTran. For the segmentation sub-model, the EC1 mechanism improves the model's accuracy from 93.85% to 100%. Through manual statistics, we find that the EC for divisible optimization accounts for the largest proportion. For the translation sub-model, the Neutron's accuracy is increased by 21.95%, reached 99.98%. From the above experimental results, We can see that the iterative EC module is critical to Neutron's performance.

Discussion

Limitations. In our work, we propose and implement a new attention-based neural decompilation framework named *Neutron*. The evaluation shows that the approach performs well. However, there still exist several limitations. Firstly, *Neutron* not effectively restores the semantics of target low-level PL, and the code comprehensibility needs to be improved. Secondly, Neutron has poor translation performance for compiler-optimized code, for we adopted the slicing mechanism, which aims to reduce the difficulty of the model and consider GPU resources' limitations. The high-level optimized code adopts a more advanced register allocation mechanism with a large front and back dependency and is challenging to perform fine-grained slicing. Thirdly, *Neutron* is powerless to identify and recover user-defined datatypes, such as classes, structures, which enables to improve the comprehensibility of the decompiled high-level PL.

Future Work. We will continue to explore techniques for improving the translation effect and semantic recovery accuracy of Neutron together with resolving the above limitations to expand Neutron's translation capabilities. For example, we will add optimized code data to the

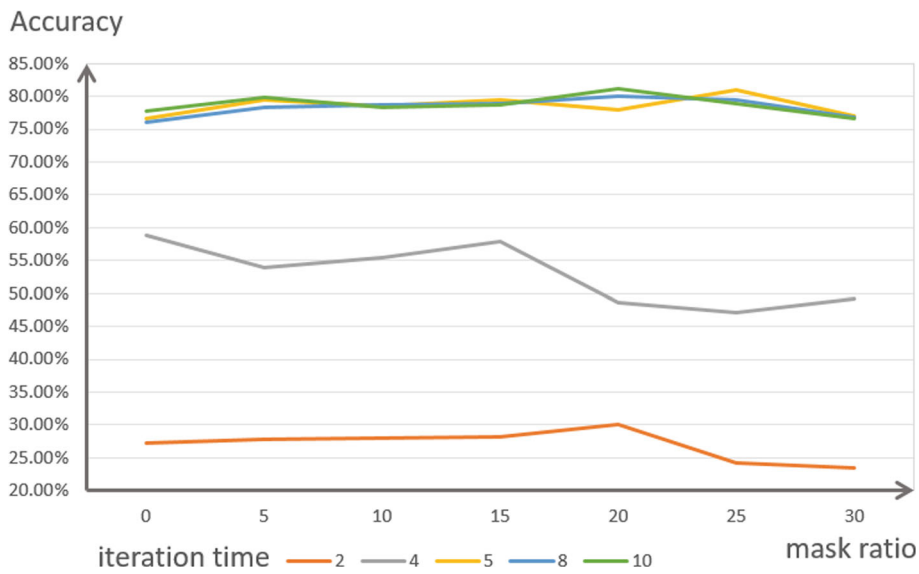

Fig. 5 Impact of iteration times and mask ratio

training dataset and try to let Neutron learn more optimization rules to decompile optimized low-level PL codes accurately. We will also learn from the existing work of identifying and restoring user-defined datatypes, further proposing a neural-based user-defined data identification approach. The method can improve the comprehensibility of decompiled high-level PL to provide technical assistance for vulnerability detection and malicious code analysis.

Conclusion

In this paper, we propose Neutron, the new decompilation architecture based on the LSTM-Seq2Seq-attention mechanism, which can accurately translate low-level PL, such as assembly code into high-level PL with similar functions. Besides, we design a novel translation mechanism based on the PL's basic operation to make the NMT model more accurate, efficiently capture the translation rules between PLs, and improve the NMT model's generalization ability. The results on three real-world projects and three different tasks show that Neutron's accuracy can reach 96.96% on average.

Authors' contributions

All authors have contributed to this manuscript and approve of this submission. Ruigang Liang and Ying Cao participated in all the work and drafting the article. Peiwei Hu has made many contributions to the technical route, designing research, and revising the article. Prof. Kai Chen made a decisive contribution to the content of research and revising the article critically.

Funding

Our research was supported by NSFC U1836211. And the recipient is Professor Kai Chen.

Availability of data and materials

We confirm that this manuscript has not been published elsewhere and is not under consideration by another journal.

Competing interests

We confirm that none of the authors have any competing interests in the manuscript.

Received: 16 November 2020 Accepted: 3 January 2021
Published online: 05 March 2021

References

Allamanis M, Tarlow D, Gordon A, Wei Y (2015) Bimodal modelling of source code and natural language. In: International Conference on Machine Learning. pp 2123–2132

Avast Retargetable Decompiler IDA Plugin (2020). https://doi.org/blog.fpmurphy.com/2017/12/avast-retargetable-decompiler-ida-plugin.html

Brumley D, Lee J, Schwartz E, Woo M (2013) Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: Presented as Part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13). pp 353–368

cfile (2020). https://doi.org/github.com/cogu/cfile

Ďurfina L, Křoustek J, Zemek P, Kolář D, Hruška T, Masařík K, Meduna A (2011) Design of an automatically generated retargetable decompiler. In: Proceedings of the 2nd International Conference on Circuits, Systems, Communications & Computers. pp 199–204

Ďurfina L, Křoustek J, Zemek P (2013) Psybot malware: A step-by-step decompilation case study. In: 2013 20th Working Conference on Reverse Engineering (WCRE). pp 449–456. IEEE

Fu C, Chen H, Liu H, Chen X, Tian Y, Koushanfar F, Zhao J (2019) Coda: An end-to-end neural program decompiler. In: Advances in Neural Information Processing Systems. pp 3703–3714

Ghidra (2020). https://doi.org/ghidra-sre.org

Heo K, Oh H, Yi K (2017) Machine-learning-guided selectively unsound static analysis. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp 519–529. IEEE

Hex-Rays (2020). https://doi.org/.hex-rays.com/products/decompiler/

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural computation 9(8):1735–1780

Karel the Robot (1995). https://doi.org/.cs.mtsu.edu/~untch/karel/

Katz DS, Ruchti J, Schulte E (2018) Using recurrent neural networks for decompilation. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp 346–356. IEEE
Katz O, Olshaker Y, Goldberg Y, Yahav E (2019) Towards neural decompilation. CoRR abs/1905.08325. https://doi.org/arxiv.org/abs/1905.08325
Křoustek J, Matula P, Zemek P (2017) RetDec: An Open-Source Machine-Code Decompiler. December 2017, technická správa, prezentované na konferencí Botconf
Levy D, Wolf L (2017) Learning to align the source code to the compiled object code. In: International Conference on Machine Learning. pp 2043–2051
Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 abs/1801.01681
Liu Z, Wang S (2020) How far we have come: testing decompilation correctness of c decompilers. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp 475–487
Loyola P, Marrese-Taylor E, Matsuo Y (2017) A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers). Association for Computational Linguistics, Vancouver. pp 287–292. https://doi.org/10.18653/v1/P17-2045. https://www.aclweb.org/anthology/P17-2045
Luong M, Pham H, Manning CD (2015) Effective Approaches to Attention-based Neural Machine Translation. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Lisbon. pp 1412–1421. https://doi.org/10.18653/v1/D15-1166. https://www.aclweb.org/anthology/D15-1166
Math c++ library (2020). https://doi.org/.cplusplus.com/reference/cmath/
Pearlmutter B (1995) Gradient calculations for dynamic recurrent neural networks: A survey. IEEE Transactions on Neural networks 6(5):1212–1228
Peng F, Deng Z, Zhang X, Xu D, Lin Z, Su Z (2014) X-force: Force-executing binary programs for security applications. In: 23rd {USENIX} Security Symposium ({USENIX} Security 14). pp 829–844
Reddy D (1977) Speech understanding systems: report of a steering committee. Artificial Intelligence 9(3):307–316. https://doi.org/10.1016/0004-3702(77)90026-1, http://www.sciencedirect.com/science/article/pii/0004370277900261
Schkufza E, Sharma R, Aiken A (2013) Stochastic superoptimization. ACM SIGARCH Computer Architecture News 41(1):305–316
Shi T, Keneshloo Y, Ramakrishnan N, Reddy C (2018) Neural abstractive text summarization with sequence-to-sequence models. arXiv preprint arXiv:1812.02303 2(1):p1–p37
Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Kruegel C, Vigna G (2016) SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy
Sutskever I, Vinyals O, Le Q (2014) Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems. pp 3104–3112
tensor2tensor (2020). https://doi.org/github.com/tensorflow/tensor2tensor
Wang Y, Zhang C, Xiang X, Zhao Z, Li W, Gong X, Liu B, Chen K, Zou W (2018) Revery: From proof-of-concept to exploitable. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp 1914–1927
Warren HS (2012) Hacker's Delight, 2nd ed. Addison-Wesley Professional
Wu Y, Schuster M, Chen Z, Le Q, Norouzi M, Macherey W, Krikun M, Cao Y, Gao Q, Macherey K, et al. (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144:11–20. https://doi.org/10.1145/3234150
Yadegari B, Johannesmeyer B, Whitely B, Debray S (2015) A generic approach to automatic deobfuscation of executable code. In: 2015 IEEE Symposium on Security and Privacy. pp 674–691. IEEE
Yakdan K, Dechand S, Gerhards-Padilla E, Smith M (2016) Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In: 2016 IEEE Symposium on Security and Privacy (SP). pp 158–177. IEEE
Yakdan K, Eschweiler S, Gerhards-Padilla E, Smith M (2015) No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In: NDSS
Yang X, Chen Y, Eide E, Regehr J (2011) Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp 283–294
Yang Z, He X, Gao J, Deng L, Smola A (2016) Stacked attention networks for image question answering. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp 21–29
You W, Zong P, Chen K, Wang X, Liao X, Bian P, Liang B (2017) Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp 2139–2154
Zong P, Lv T, Wang D, Deng Z, Liang R, Chen K (2020) Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: 29th {USENIX} Security Symposium ({USENIX} Security 20). pp 2255–2269

Publisher's Note