概 要

逆コンパイラはコンパイル後のバイナリデータからソースコードを復元するためのツールである。様々な逆コンパイラが存在するが、既存の逆コンパイラはしばしば人間にとって分かりにくいソースコードを出力する。そこで本論文では、統計的機械翻訳の技術を逆コンパイラに用いた、解析者にとってわかりやすいソースコードを出力する逆コンパイラを提案する。具体的には、機械翻訳において有用とされる注意機構付き再帰型ニューラルネットワークを用いる。実験では、オープンソースプロジェクトから収集したソースコードとそのバイナリデータを用いてニューラルネットワークを学習し、その性能を検証した。実験の結果、注意機構を用いることににより逆コンパイル結果における若干の精度向上が認められたが、実用的な精度までは至らなかった。

ニューラルネットを用いた逆コンパイラとその評価

February 1, 2018

*Acknowledgement I thank Prof. Sugiyama and Lecture Sato for providing useful advice, supplying computational resources, helping me with writing this thesis and correcting my English. I also thank A. Sano, who is my college classmate, for checking my English.

目 次

1 Decompilation with Machine Learning **4**

2 Improvement of decompilation results **4**

3 Neural Network for Program Generation **5**

4 Decompiler **6**

5 Problem Settings and the Approaches **8**
5.1 RNN and LSTM 8
5.2 Sequence-to-Sequence 9
5.3 Attention 11
5.4 Sequence-to-Tree 11

6 Data Collection **13**

7 Data Preprocessing **15**
7.1 Assembly Code Preprocessing 15
7.2 C Code Preprocessing 16

8 Model Configuration **17**

9 Model Training **17**

10 Evaluation Metrics **18**

11 Numerical Evaluation **18**

12 Sampled Decompilation Results **21**

13 Conclusion **21**

14 Future Work **23**

14.1 Improvement on Assembly Code Preprocessing 23

14.2 Improvement on Sequence-to-tree Model 23

14.2.1 Copy Leaf Data from Assembler 23

14.2.2 Path Estimation 24

14.3 Improvement on Datasets 24

Introduction

These days, the amount of malware has been increasing rapidly. In the report of AV-TEST Institute, over 100 million malwares are newly found in every year for the past five years [12]. There are many case studies for malwareinfection [14, 15, 20, 24]. In those cases, they analyse malwares to assess the impact of the incident and to identify the vulnerability which causes the infection. In order to analyse malwares quickly and efficiently, there are some tools for analysing malwares [8]. One of these tools is a decompiler, which is a tool for analyzing the behavior of malware. This tool generates a C-like pseudocode which represents the behavior of the malware, and analysts can infer the behavior of the malware from the pseudocode. This is much easier compared to analyzing raw binary codes.

However, sometimes a decompiler generates an unstructured code, which requires users to make an effort to understand the behavior of the program. This is due to flexibility of the high level language. If the decompiler chooses a human-friendly representation of the language as a result of decompiling, we can analyze malware with less effort.

In this thesis, we apply neural networks to the problem of decompilation to generate human-friendly pseudocodes. This idea is originated from Katz et al. [17], in which they used recurrent neural networks, in particular sequence-to-sequence model, for decompilation. However, their model does not seem to be implemented some features for improving the quality of decompilation. Therefore, we implemented the atten-

tion mechanism, which is introduced in Luong et al. [22], for improving the quality. Additionally, the sequence-to-sequence model, which they used for the modeling, ignores the structure of a C language code. Therefore, we also implemented the sequence-to-tree model, which is suggested in Yin and Neubig [32], for generating structured C language source code.

In the experiment, we tested three models, the basic sequence-to-sequence model, the sequence-to-sequence model with attention and the sequence-to-tree model. As the result of the experiment, the attention mechanism showed little improvement in the performance, while the sequence-to-tree does not. Unfortunately, the results of the all models were not still good enough for practical use. The results of decompilation for short C source code fragment were not so bad. Nevertheless, the results for long C codes, which are considered to appear in practical use, were mostly failed.

Related Works In this chapter, we review related works.

1 Decompilation with Machine Learning

Because perfect decompilation is naturally impossible, there are some machine learning applications for decompilation.

Katz et al. [17] used the seqence-to-sequence model, which is a popular technique in the machine translation. The seqence-to-sequence model is also evaluated in this thesis.

Schulte et al. [26] applied genetic programming to decompilation. They prepared a big corpus of C source codes, then they repeatedly mutated C codes to reduce the difference of the target binary and the compilation result of a C code.

2 Improvement of decompilation results

In this thesis, we do not try to recover the function name and variable name. However, for hand-decompilation, sometimes the name can be estimated from the functionality of the variable. For example, Van Emmerik and Waddington [30] used the Boomerang decompiler and repeatedly fixed variable types and names of the decompilation results to recover the source code of a software.

Here, we show a example of the name estimation. Assume there is a variable X which seems to have a C struct type and has two fields P and Q. Also, there are two functions F and G which operate with X. The function F receives X and another value V, assigns V to the memory indicated by P, adds 8 to P and increases Q. The function G receives X, subtracts 8 from P, decreases Q, and returns the value indicated by P. In the situation, is is estimated that X,F and G have names related to *stuck*, *push* and *pop*, respectively.

Jaffe et al. [13] tackled this work. In that paper, they tried to estimate the name of a variable which appears in the decompilation result of a compiled C program.

This kind of name estimation also appears in the context of deobfuscation. [29] Deobfuscation is a reverse process of obfuscation. Obfuscation is a technique to modify the program source code so that the code becomes hard to understand for humans. Obfuscation is used for hiding the behavior of a program whose source code has to be distributed. For example, in the web application, a JavaScript code of a program should be distributed in order to be executed on client web browsers. Such JavaScript code is minified in order to protect the program from cracking. All comments, newlines, indents are removed and variable names are substituted to automatically generated meaningless names.

Vasilescu et al. [31] applied statistical machine translation for this task. They calculated the future value of a variable from its functionality, then sought the most suitable name from the candidates generated from the corpus.

3 Neural Network for Program Generation

There are many applications of neural networks for program synthesis. They are summarized by Kant [16].

In this thesis, we applied sequence-to-tree model suggested by Yin and Neubig [32], because the input of the decompiler is a sequence of machine codes and the output of the decompiler is an AST.

For another example, Chen et al. [7] tried to mutually translate from JavaScript to CoffeeScript, and its inverse. In that case, both JavaScript codes and CoffeeScript codes are tree structured, so they suggested tree-to-tree model.

Problem Settings and Methods

In this chapter, we explain a decompiler, its statistical formalization and the neural network models used in the experiment.

4 Decompiler

A compiler is a program which compiles a source code written in language Y to another language X. For example, a GNU C compiler translates the C language to the x64 assembly language, a javac compiler translates Java to a Java virtual machine code. A decompiler is a program which aims to reverse the process of a compiler, retrieving a source code of the high-level language Y from a source code of a low-level language X. When Y is a high-level language and X is relatively a low-level language, this reverse process becomes insufficient or impossible [11, 23]. This is because a lot of information, such as variable names or function names, is lost when compiling. Additionally, the program structure described in the high level-language is lost too. For example, a simple `for` statement can be represented by the combination of a `while` statement and an `if` statement, or the combination of an `if` statement and a `goto` statement. So a decompiler cannot distinguish the difference in their representations, although in the real situation `for` statements are often used and `goto` statements are rarely used.

Figure 1 shows an example of decompilation results of the Snowman decompiler version v0.1.3 and the REC Studio decompiler version 4.1 for a sample C language code. The sample C code is compiled by the GNU C Compiler version 4.8.5, with the optimization level `-O0`. The REC decompiler missed the structure of the variable `root`. It also failed to estimate the number of arguments and the return value of the function. The Snowman Decompiler succeeded in reconstructing the structure of the variable as `struct s0`. It also succeeded in estimating the number of arguments and the return value as semantically correct expressions. However the statement `return 0;` is reconstructed to a complex combination of statements.

These incorrect or verbose results might be due to missing patterns in the assembly codes. Existing decompilers are made up with many steps similarly to compilers [6, 11]. In each step, a decompiler finds patterns in binary data, converts them to more high-level structure. However, covering all patterns might be hardly possible, since there are many compilers and compiler optimizations [11].

Sample C source code

```
struct s0 {
    int32_t f0;
    signed char[4] pad8;
    struct s0* f8;
};
struct s0* get_node(struct
     s0* rdi, int32_t esi) {
    struct s0* v3;
    int32_t v4;
    struct s0* rax5;
    v3 = rdi;
    v4 = esi;
    while (v3) {
        if (v3->f0 == v4)
            goto
                addr_400545_4
                ;
        v3 = v3->f8;
    }
    *reinterpret_cast<
        int32_t*>(&rax5)
        = 0;
    *reinterpret_cast<
        int32_t*>(
        reinterpret_cast<
        int64_t>(&rax5) +
         4) = 0;
    addr_400563_7:
    return rax5;
    addr_400545_4:
    rax5 = v3;
    goto addr_400563_7;
}
```

Result of the Snowman Decompiler

Result of the REC Studio Decompiler

図 1: Decompilation results of existing decompilers for C source code

By utilizing source code corpora, a statistical method might find patterns automatically, without enumerating each pattern. Therefore, we will apply some statistical methods, which are described in the next section.

5 Problem Settings and the Approaches

In this thesis, we try to apply a statistical approach, specifically a statistical machine translation (SMT) technique, for decompilation. SMT is the task which tries to translate one sentence y written in one language into the sentence x written in another language.

The decompilation problem is formulated as follows. Suppose that there is a compiler which converts a source code y written in one programming language to a source code x written in another programming language with probability $p(x|y)$. In addition, suppose that y is a human-generated source code and follows the probability distribution $p(y)$. In this situation, the decompilation probability distribution $p(y|x)$ is in proportion to $p(x|y)p(y)$ according to the Bayes law, and $\arg\max_y p(y|x)$ is considered as the best human-intelligible decompilation result for x.

In the experiment, we collected source codes from open source projects, which are regarded as the random sampling from the distribution $p(y)$, and compiled them to generate the data pairs $\{(x_k, y_k)\}_k$. Then the data pairs are used for training the SMT models to estimate $p(y|x)$, and testing the performance of the models.

5.1 RNN and LSTM

A recurrent neural network (RNN) is a neural network for dealing with sequence data. The structure of the RNN is the sequence of the same node network units, as shown in Figure 2 and Figure 3. The long short-term memory (LSTM) is a unit of the RNN which we used in the experiment. It was first introduced by Hochreiter and Schmidhuber [10] , for overcoming the vanishing gradient problem and the exploding gradient problem. In short, an LSTM unit receives two input vectors, one is the internal state of the LSTM unit, and the other is the input for the LSTM unit, and the LSTM unit emits one output vector, which represents the next internal state of the LSTM unit.

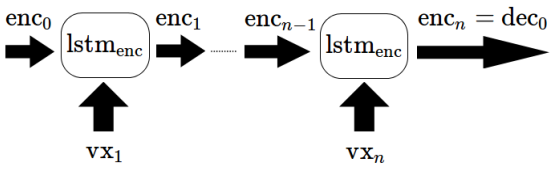

図 2: The encoder network of the sequence-to-sequence model

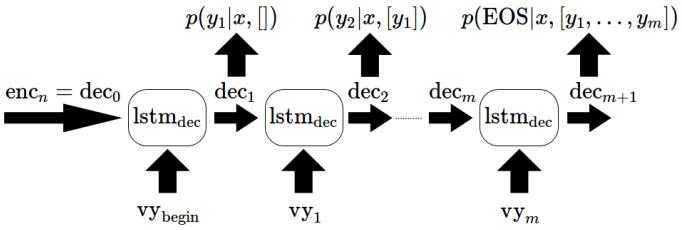

図 3: The decoder network of the sequence-to-sequence model

5.2 Sequence-to-Sequence

The sequence-to-sequence model is the model for SMT task, introduced by Sutskever et al. [28]. The sequence-to-sequence model is composed of two recurrent neural networks, the encoder network and the decoder network.

Figure 2 shows the structure of the encoder network. The purpose of the encoder network is to summarize the input sequence x, whose length varies depending on the sequence x, into a vector of fixed size. This process is carried out as follows. First, x is tokenized to sequence $x_1 \ldots x_n$, and each token x_k is embedded to its vector representation vx_k. Then, the initial LSTM internal state enc_0 is initialized randomly, and $\mathrm{enc}_1 \ldots \mathrm{enc}_n$ are calculated in accordance with the recursive formula $\mathrm{enc}_k = \mathrm{lstm}_{\mathrm{enc}}(\mathrm{enc}_{k-1}, \mathrm{vx}_k)$. Finally, the last internal state enc_n is returned as the reduced representation of the input sequence x.

Figure 3 shows the structure of the decoder network. The purpose of the decoder network is to estimate the distribution $p(y_k|x,[y_1,\ldots,y_{k-1}])$ for each k. The training process is carried out as follows. First, y

is tokenized, embedded to a vector representation $\mathrm{vy}_1 \dots \mathrm{vy}_m$ in the same way as embedding x. Then, the first internal state dec_1 is calculated from the start vector $\mathrm{vy}_{\mathrm{begin}}$ and encoded input $\mathrm{enc}_n = \mathrm{dec}_0$ in accordance with the formula $\mathrm{dec}_1 = \mathrm{lstm}_{\mathrm{dec}}(\mathrm{dec}_0, \mathrm{vy}_{\mathrm{begin}})$. Then, $\mathrm{dec}_2 \dots \mathrm{dec}_{m+1}$ are calculated in accordance with the recursive formula $\mathrm{dec}_{k+1} = \mathrm{lstm}_{\mathrm{dec}}(\mathrm{dec}_k, \mathrm{vy}_k)$. Each decoder internal state dec_k is passed to the linear layer W_{dec} and then softmaxed to generate a probability vector $\mathrm{py}_k = \mathrm{softmax}(W_{\mathrm{dec}}(\mathrm{dec}_k))$. The probability vector py_k is intended to represent the probability distribution $p(y_k|x, [y_1, \dots, y_{k-1}])$, and last probability vector py_{m+1} is intended to represent the probability of the end of the sequence $p(\mathrm{EOS}|x, [y_1, \dots, y_m])$. Thus the network is trained to reduce the difference between predicted distribution py_k and actual distribution $p(y_k|x, [y_1, \dots, y_{k-1}])$. In the experiment, we chose cross-entropy of py_k and $p(y_k|x, [y_1, \dots, y_{k-1}])$ as a loss function.

The inference from sequence x is carried out by estimating $\arg\max_y p(y|x) = \arg\max_y \prod_{k=1}^{n} p(y_k|x, [y_1, \dots, y_{k-1}])$. Calculating argmax directly is exponentially time consuming, therefore we have to approximate the argmax by calculating $y = [y_1, \dots, y_m]$ sequentially. First, y_1 is calculated as $\arg\max_{y_1} p(y_1|x, [])$, where the distribution $p(y_1|x, [])$ is approximated by $\mathrm{softmax}(W_{\mathrm{dec}}(\mathrm{dec}_1))$. Then, the next token $y_2 = \arg\max_{y_2} p(y_2|x, [y_1])$ is calculated from dec_1 and the vector embedding of y_1, and so on. When the token y_{m+1} is the EOS token, the generating process is stopped and the sequence $[y_1, \dots, y_m]$ is returned as an inference result.

What is mentioned above is the basic structure of the sequence-to-sequence model. To improve the performance of the sequence-to-sequence model, there are some devices for the model. In the following, we introduce two devices, which are used in our experiment.

The first is the bidirectional RNN (BRNN), which is the architecture introduced by Schuster and Paliwal [27]. In our experiment, this architecture is used for the encoder network as follows. In addition to the internal state enc_k, the reverse internal state $\mathrm{enc}_k^{\leftarrow}$ is also calculated in accordance with the recursive formula $\mathrm{enc}_k^{\leftarrow} = \mathrm{lstm}_{\mathrm{enc}}^{\leftarrow}(\mathrm{enc}_{k+1}^{\leftarrow}, \mathrm{vx}_k)$. Then both of the internal states enc_n and $\mathrm{enc}_0^{\leftarrow}$ are used for the decoder network.

The second is the multi-layer RNN, introduced by Graves et al. [9]. They reported that, feeding the output sequence of one LSTM network to another LSTM network, similarly to stacking the network layers,

will improve the accuracy of prediction. This architecture is called the stacked RNN or the multi-layer RNN.

5.3 Attention

Attention is an extension module for the sequence-to-sequence model to improve the quality of translation. A simple sequence-to-sequence model has a problem that the size of the encoded vector enc_n is fixed regardless of the length of the input sequence x. Therefore, the longer the length of the input becomes, the more likely the information of the input will be lost. This problem is eased by using the information in the encoding network when decoding the output sequence.

The attention module was first developed by Bahdanau et al. [5]. We implemented the global general attention, which was introduced by Luong et al. [22], for improving the quality of decompilation. In the following, we will describe the global general attention.

As the mentioned in the previous section, the pure sequence-to-sequence model uses the output of the decoder network dec_k for estimating the distribution $p(y_k|x, [y_1, \ldots, y_{k-1}])$. In the sequence-to-sequence model with attention, a context vector ctx_k is calculated from dec_k and the internal states of encoder networks $\mathrm{enc}_1 \ldots \mathrm{enc}_n$, then the distribution is estimated from the output vector dec_k and context vector ctx_k. The context vector ctx_k is calculated as follows. For each encoder state enc_l, the importance of the state is calculated as $a_l = \mathrm{score}(\mathrm{dec}_k, \mathrm{enc}_l)$, then ctx_k is calculated as $\mathrm{ctx}_k = \sum_{l=1}^{n} \frac{\exp a_l}{\sum_{l=1}^{n} \exp a_l} \mathrm{enc}_l$, as the weighted average of $\mathrm{enc}_1 \ldots \mathrm{enc}_n$. In Luong et al. [22], various kinds of score functions were suggested. In our experiment, we used a general score function, which is calculated as $\mathrm{score}(\mathrm{dec}_k, \mathrm{enc}_l) = \mathrm{dec}_k^T W \mathrm{enc}_l$, where W is a parameter matrix.

5.4 Sequence-to-Tree

Compared to natural languages, programming languages obey rigid grammars. Therefore, it is reasonable to consider the grammar of the programming language when generating programs. Sequence-to-tree is a model for generating a tree-structured output from a sequence input. This model was introduced by Yin and Neubig [32] for generating a program from a program specification written in natural language.

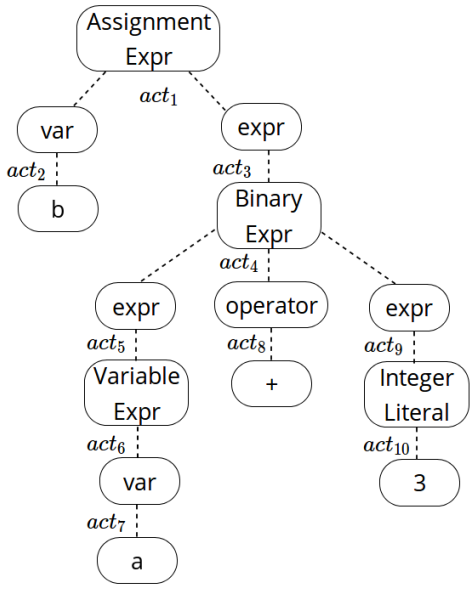

図 4: abstract syntax tree (AST) representation of a C statement `b = a + 3;`

This task is similar to our task which aims to generate programs from assembly codes, which are sequences of instructions. Therefore we implemented this model in the experiment.

Now, we will describe the sequence-to-tree model. The sequence-to-tree model is composed of two networks, the encoder network and the decoder network. The encoder network of the sequence-to-tree model is the same as that of the sequence-to-seequence model. It summarizes an input sequence into a vector of fixed size.

The decoder network is a network for generating tree structured output. A tree structure is representable by a sequence of the actions which expands a nonterminal symbol of the constructing tree. For example, the tree shown in Figure 4 is generated from the sequence $[\mathrm{act}_1, \ldots, \mathrm{act}_{10}]$, where action act_k ex-

pands the left-most nonterminal symbol in the tree generated from the action sequence $[\mathrm{act}_1, \ldots, \mathrm{act}_{k-1}]$. Suppose the tree t is generated by the action sequence $[\mathrm{act}_1, \ldots, \mathrm{act}_m]$, then the tree-generating probability $p(t|x)$ is decomposed into $\prod_{k=1}^{n} p(\mathrm{act}_k|x, [\mathrm{act}_1, \ldots, \mathrm{act}_{k-1}])$. Thus the sequence-to-sequence model is now applicable to generating tree structure, by embedding each action act_k to represent vector av_k and calculate $\mathrm{lstm}_{\mathrm{dec}}(\mathrm{dec}_k, \mathrm{av}_k)$ sequentially for estimating $p(\mathrm{act}_{k+1}|x, [\mathrm{act}_1, \ldots, \mathrm{act}_k])$.

In the sequence-to-tree model, the information which is peculiar to the tree structure is additionaly used. For expanding node d, the information of the node type nf_d and the parent action pa_d from which node d is generated is used. Then the next internal state is calculated as $\mathrm{lstm}_{\mathrm{dec}}(\mathrm{dec}_k, [\mathrm{av}_k; \mathrm{nfv}_d; \mathrm{pav}_d])$ instead of $\mathrm{lstm}_{\mathrm{dec}}(\mathrm{dec}_k, \mathrm{av}_k)$, where $\mathrm{nfv}_d, \mathrm{pav}_d$ are the vector embeddings of $\mathrm{nf}_d, \mathrm{pa}_d$.

In Yin and Neubig [32], they also used a devised method for generating terminal symbols, because they need to copy the name of some variables from the program specification. However, in our case, we do not need to copy the information from the input, because the name information is lost when compiling. Nevertheless, it was found from our preliminary experiments that our models often fail to recover the constant values in the programs. Therefore, implementing the copying method for values is left as future work.

Evaluation Experiment In this chapter, we explain the details of the evaluation experiment. The source codes for the experiment are available at `https://github.com/satos---jp/neural_decompiler`.

6 Data Collection

We collected source codes from GitHub [3] by crawling. We cloned about 900 repositories with the C language topic, and recursively searched each repositoriy to enumerate C language source codes. From each source code, we generated pairs of a C source code fragment and an x64 assembly code corresponding to the C source code fragment. Some example data pairs are shown in Figure 5.

These pairs were generated as follows. Let S0 be an original C source code.

1. Remove all preprocessors, such as `#include`, in S0 to generate preprocessed C code S1.

C source code	x64 assembly
r *= a	mov eax,DWORD [rbp−0x4] imul eax,DWORD [rbp−0x14] mov DWORD [rbp−0x4],eax

while(p >= 0){
p−−;
r *= a;
}
	jmp LABEL_0
	LABEL_1 :
	sub DWORD [rbp−0x18],0x1
	mov eax,DWORD [rbp−0x4]
	imul eax,DWORD [rbp−0x14]
	mov DWORD [rbp−0x4],eax
	LABEL_0 :
	cmp DWORD [rbp−0x18],0x0
	jns LABEL_1

int pow(**int** a,**int** p){
int r = 1;
while(p >= 0){
p−−;
r *= a;
}
return r;
}
	push rbp
	mov rbp,rsp
	mov DWORD [rbp−0x14],edi
	mov DWORD [rbp−0x18],esi
	mov DWORD [rbp−0x4],0x1
	jmp LABEL_0
	LABEL_1 :
	sub DWORD [rbp−0x18],0x1
	mov eax,DWORD [rbp−0x4]
	imul eax,DWORD [rbp−0x14]
	mov DWORD [rbp−0x4],eax
	LABEL_0 :
	cmp DWORD [rbp−0x18],0x0
	jns LABEL_1
	mov eax,DWORD [rbp−0x4]
	pop rbp
	ret

図 5: Example of training data pairs

2. Tokenize S1 and rearrange into S2 so that there is exactly one token for each line.

3. Compile S2 by the GNU C compiler with no optimization option (-O0) and debug option (-g) to get assembly code S3. With the debug option, S3 contains the information about the corresponding source code positions in S2.

4. Assemble S3 to object file S4 and disassemble S4 to get information eliminated assembly code S5.

5. Parse S2 with a Clang compiler to get the abstract syntax tree (AST) of the source code and enumerate the subtrees of the AST. Each subtree of the AST has the corresponding source code positions in S2. Therefore, we can get a pair of an assembly code and a C fragment corresponding to the subtree by using the informations in S2, S3 and S5.

The pairs of a C fragment and an assembly code are then preprocessed for the training and the testing data.

7 Data Preprocessing

In order to apply the sequence-to-sequence model, we have to preprocess the assembly codes and the C fragments into the sequences of tokens.

7.1 Assembly Code Preprocessing

There are two ways for preprocessing assembly codes.

The first way is converting each code into the sequence of integers. In order to execute as a program, an assembly code is assembled into the sequence of one-byte integers, ranging from 0 to 255. Therefore the one-byte integer sequences can be directly used as input sequences to the sequence-to-sequence model.

The second way is converting each code into the tokenized sequence of assembly codes. That is, we split the disassembled instructions in S5 into the sequence of symbols, concatenate these sequences to one sequence, and modify it as follows:

- For each end of the instructions, the `ENDLINE` token is added to indicate the end of the instructions.
- The relative addressing, such as the address of the `call` or `jump` instructions, is normalized by renaming so as to make the code position independent and to reduce the vocabulary size.
- Big integer values are clipped. More specifically, an integer in the range from -255 to 256 are preserved as it is, and other integers are clipped and substituted with the `__HEX__` symbol. This also reduces the vocabulary size.

We chose the second way for the experiment. This is because, in our case, the assembly code is generated from the GNU C compiler, which is a common compiler. Therefore the assembly code can be disassembled uniquely. For example, Linn and Debray [21] proposed an obfuscation technique, which deceives a disassembler to generate wrong disassembly results. In that case, the first way might be better than the second way.

7.2 C Code Preprocessing

The C source codes were parsed to AST by a Clang compiler [2] and its Python bindings together with pycparser [4], because all of the tools are individually not sufficient for generating a dataset. Then, the AST is converted to input data as follows:

- The variable-ary nodes and optional nodes were converted. For example, the `CompoundStatement` node has a variable number of statements. We converted such a list of nodes to the chains of the node `CONS` and the node `NIL`, as the list representation in a functional language. Another example is the `if` statement, whose `else` clause is sometimes missing. We converted such optional nodes to the alternative of the `SOME` node and the `NONE` node, as the optional data structure in a functional language.
- The variable names, function names, label names, or type names were renamed similarly to the addresses in the assembly code data.
- There are four type of literals in the C language. An integer literal less than 256 was preserved, and an integer literal greater than 256 was clipped to 256. The string, float and character literals were converted to the `__LITERAL__` symbol.

図 6: The configuration parameters of the models

	Seq2Seq	Seq2SeqAtt	Seq2Tree
Generate	token sequence		AST
With Attention	no	yes	
Encoder	Bidirectional LSTM		
Number of LSTM Layers	4		
Dimensionality of Vector Embedding	128		

ASTs were used for the sequence-to-tree model as they were, or converted to C token sequences for the sequence-to-sequence model.

8 Model Configuration

The training model was implemented with Chainer version 4.5.0 [1]. We implemented three models, Seq2Seq, Seq2SeqAtt, and Seq2Tree.

The encoder network was a bidirectional LSTM in all of the three models. The decoder network of Seq2Seq and Seq2SeqAtt predict the conditional probability of the next token $p(y_k|x, [y_1, \ldots, y_{k-1}])$, while the decoder network of Seq2Tree predicts the conditional probability of the next action $p(\mathrm{act}_k|x, [\mathrm{act}_1, \ldots, \mathrm{act}_{k-1}])$.

Seq2SeqAtt and Seq2Tree are the models with attention, while Seq2Seq is not.

Figure 6 shows the parameters of the models.

9 Model Training

We trained these models to reduce the cross-entropy loss between predicted and actual conditional probabilities. The conditional probability is the probability of next token $p(y_k|x, [y_1, \ldots, y_{k-1}])$ for the sequence-to-sequence model, and the probability of next action $p(\mathrm{act}_{k+1}|x, [\mathrm{act}_1, \ldots, \mathrm{act}_k])$ for the sequence-to-tree model.

For the optimizer of the training, we used Adam, which was introduced by Kingma and Ba [18]. The parameters of Adam were set to $(\alpha, \beta_1, \beta_2) = (0.001, 0.9, 0.999)$, which are the default parameters of Chainer version 4.5.0.

10 Evaluation Metrics

We evaluated the quality of the results by two metrics, the edit distance ratio [17] and the bilingual evaluation understudy (BLEU) [25].

The edit distance ratio is a metric for for the correctness of the decompilation result. Here, the edit distance ratio between two strings means the Levenshtein distance [19] divided by the length of the longer string. Ideally, the correctness of the decompilation result should be measured by the distance of the meaning of the program. However the decompilation result is usually uncompilable, thus we used the edit distance of the source codes as an approximation of the correctness.

BLEU is the metrics for the similarity of the ground truth and the decompilation result. It was first introduced by Papineni et al. [25], for evaluating the performance of machine translation. BLEU is calculated from the sets of n-grams for each sentence. If the two sentences have similar n-gram sets, the BLEU score becomes higher. In our case, it means that the decompilation result is more similar to the program written by human.

Before calculating these scores, we normalized variable names in the source codes by renaming.

11 Numerical Evaluation

In order to evaluate each model, we randomly sampled 100 data pairs from test cases. For each test case, we generated decompilation results from the assembly code data of the test case. Then, we averaged the edit distance ratios and BLEU scores between results and ground truths.

Figure 7 shows the relation between the training data size and the averaged edit distance ratio for each model. Figure 8 shows the relation between the training data size and the averaged BLEU score for each model. The lower edit distance ratio and the higher BLEU score mean better decompilation performance.

We can see that Seq2SeqAtt shows slightly better results in both the edit distance ratios and BLEU scores, which implies that the attention improves the performance of the decompilation. However, there is no significant difference between Seq2Seq and Seq2Tree in both metrics. This seems to be due to the length of the sequences to be estimated.

Figure 9 shows the length distribution of the dataset as C token sequences, and Figure 10 shows the size distribution of the same dataset as

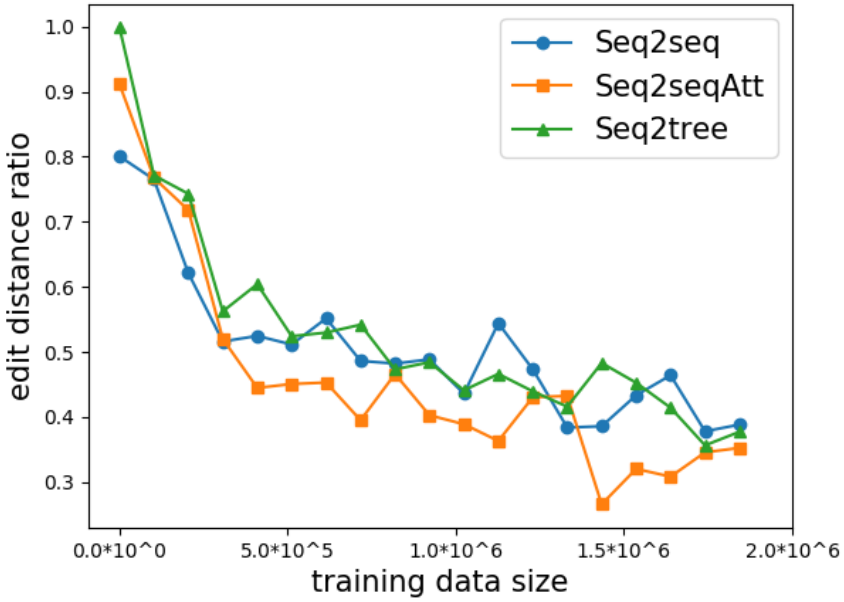

図 7: Averaged edit distance ratio for each models

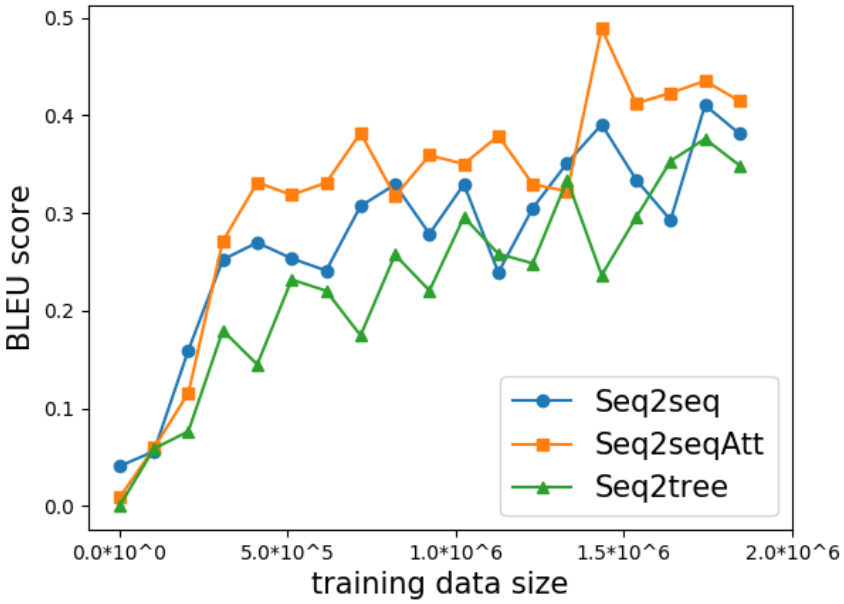

図 8: Averaged BLEU score for each models

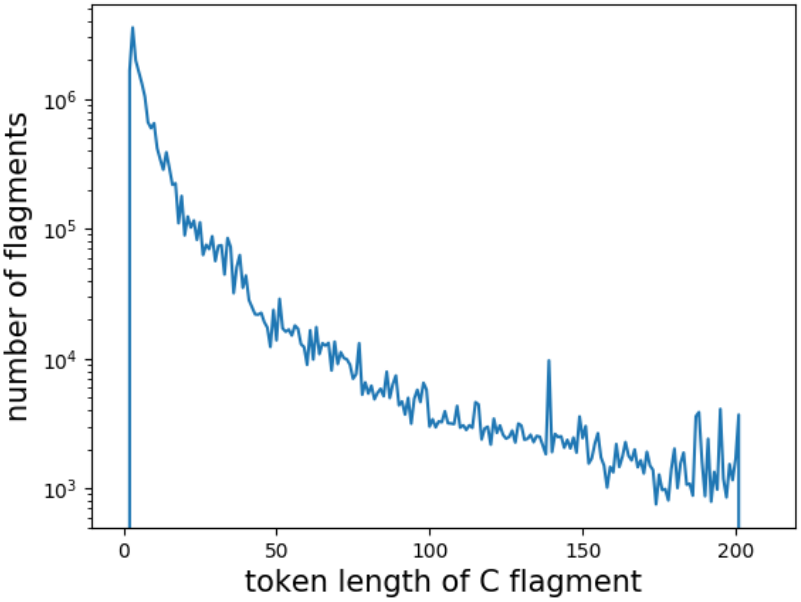

図 9: Length distribution of C token fragments

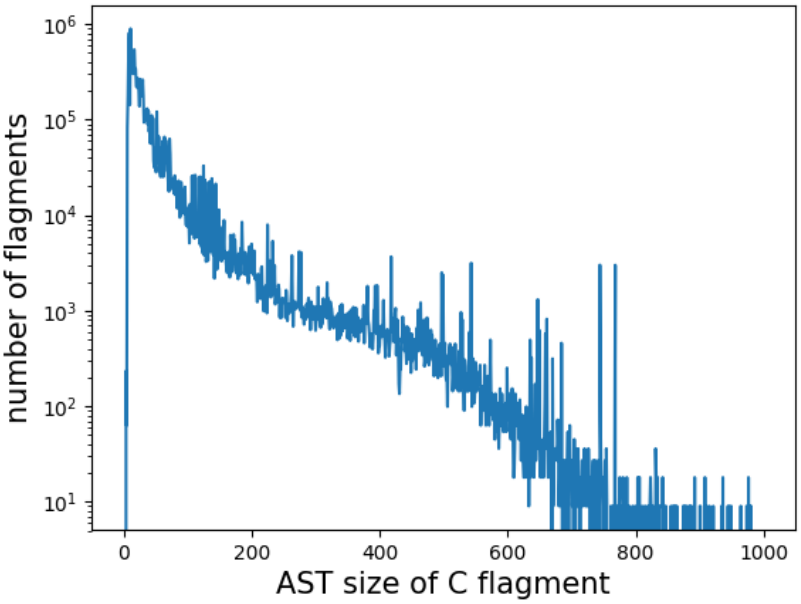

図 10: Length distribution of datasets as AST

ASTs. The size of an AST is equal to the length of its action sequence. According to these two figures, the AST size is 2 or 3 times larger than the length of C token sequences. Therefore, the AST estimation may be harder than the C token sequence estimation.

12 Sampled Decompilation Results

Figure 11 shows the sample translate results for each model.

For the first short example, each of the models recovered that the C statement is composed of multiplication and assignment, while only Seq2Tree recovered the name of the variables correctly, and Seq2Seq and Seq2SeqAtt were failed to recover.

For the second middle example, Seq2Seq outputted source code with invalid parentheses, Seq2SeqAtt outputted invalid syntax code (combination of the `while` statement and the `else` statement). Seq2Tree outputted a correct syntax of codes and estimated the `while` loop correctly, but failed to estimate subexpressions.

For the third large example, all of the three models outputted a function with wrong body statements. The `INVALID_FUNCTION_DECL` in the result of Seq2Tree means it outputted a wrong structured function type, and the `TRANCATE` means the action was truncated because the estimated action sequence was too long. For long size functions or statements, Seq2Tree tends to expand the list type node to the `CONS` node, which causes infinite expansion of a list, and as a result, the type of the output function becomes broken and the body of the function becomes truncated.

According to these decompilation results, it can be said that these models are not acceptable for practical use.

Conclusion and Future Work

13 Conclusion

We tested three neural network models Seq2Seq, Seq2SeqAtt and Seq2Tree for decompilation. Numerically, Seq2SeqAtt performed better compared to the other two models, but all of the three models were not still good enough for practical use. Therefore, we will continue to develop a practical decompiler for the good future of malware analysis.

ground truth	r ∗= a
Seq2Seq	**const unsigned int** V0 = V1 ∗ floatliteral
Seq2SeqAtt	V0 = (**unsigned int**) (V1) ∗ (**unsigned int**) V2
Seq2Tree	V0 = V0 ∗ V1

ground truth	**while**(p >= 0){
	p−−;
	r ∗= a;
	}

Seq2Seq	**for**(;V0 = (1<<(((**unsigned int**)−1<<((32−20)−1))−1)) ∗((**unsigned char**∗)(V1.V2))[1] = (**unsigned char**)2; }

Seq2SeqAtt	**while**(((**void** ∗) >= 0) != ((**void** ∗) 0)){
	return ((**void** ∗) 0);
	} **else** {
	V0 = ((**void** ∗) 0);
	}

Seq2Tree	**while**(V0 < 0){
	V1 = V0 − 1;
	}

ground truth	**int** pow(**int** a,**int** p){
	int r = 1;
	while(p >= 0){
	p−−;
	r ∗= a;
	}
	return r;
	}

Seq2Seq	**void** V0(**int** V1,**int** V1){
	int V1 = 0;
	int V1;
	for(V2=1;V1<=V2;++V2){
	V1 = ∗ V3;
	}
	}

Seq2SeqAtt	**void** V0(**unsigned int** V1,**unsigned int** V2){
	unsigned int V1 = ((**void** ∗) 0);
	unsigned int V3 = ((**void** ∗) 0);
	unsigned int V1 = ((((**void** ∗) 0) != ((**void** ∗) 0)));
	return V4;
	}

Seq2Tree	INVALID_FUNCTION_DECL {
	TRANCATETRANCATE
	}

図 11: Sample translation results for each model

14 Future Work

In this section, we suggest some other approaches to improve the accuracy of our decompilation models.

14.1 Improvement on Assembly Code Preprocessing

We used tokenized assembly string sequences for training. However, such simple concatenation of sequences might lose the information of the unity of each instruction. To improve this point, we propose the following method.

This method uses two LSTM networks, the instruction encoder network and the assembly code encoder network, for encoding assembly token sequences. First, each assembly instruction sequence is encoded by the instruction encoding network into one instruction embedding vector, thus the assembly codes turn into the sequences of instruction embedding vectors. Then, the seqeuence of embedding vectors is encoded into one reduced vector by the assembly code encoder network.

This method will be time consuming compared to the method used in the experiment. Nevertheless, this might be worth trying.

14.2 Improvement on Sequence-to-tree Model

We considered that the AST representation could exploit the feature of C source codes. However, the sequence-to-tree model was not well performed. In this section, we seek to the improvement of the AST approach.

14.2.1 Copy Leaf Data from Assembler

The translation results show that the sequence-to-tree model has difficulty in recovering the leaf data of the program. Therefore changing the method for expanding leaf nodes might improve the accuracy of decompilation. There are three kinds of leaf data in the AST representation of a C source code.

The first kind is the literal data. In the C source code, there are four literal data, integer, float, char and string. In the assembly code, the string literal appears in the form of a memory address, and the

other three literals appear in the form of an immediate value. Therefore, instead of estimating the literal value directly, estimating the corresponding position in the assembly code might improve the correctness of decompilation.

The second kind is the type of operators. They can be also estimated from the assembly code. For example, the operator + is compiled to the opcode `add`, the operator | is compiled to the opcode `or`, and so on. Thus the type of the operator might be estimated by indicating the correspond position in the assembly code.

The third kind is the variable name. In the assembly code, a variable corresponds to a memory access. Therefore, the name of a variable is identified by its corresponding memory position, similarly to the above two kinds.

14.2.2 Path Estimation

In the sequence-to-tree model, a tree structure is flattened into an action sequence, therefore the information of a tree structure might be lost in some degree. For example, adjacent actions in the action sequence might be distant actions in an AST, when they are in the different subtree of the AST.

In order to preserve the information of the tree structure, we suggest to train the decoder network for estimating each root-to-leaf action sequence in the AST, instead of the flattened action sequence. For example, in Figure 4, there are four root-to-leaf action sequences, $[\mathrm{act}_1, \mathrm{act}_2]$, $[\mathrm{act}_1, \mathrm{act}_3, \mathrm{act}_4, \mathrm{act}_5, \mathrm{act}_6, \mathrm{act}_7]$, $[\mathrm{act}_1, \mathrm{act}_3, \mathrm{act}_4, \mathrm{act}_8]$ and $[\mathrm{act}_1, \mathrm{act}_3, \mathrm{act}_4, \mathrm{act}_9, \mathrm{act}_{10}]$. The AST can be recovered from these sequences instead of the whole sequence $[\mathrm{act}_1 \ldots \mathrm{act}_{10}]$, and these sequences are independent of each other, when generating the AST. Therefore, we may estimate each of the sequences from the assembly token sequence by the sequence-to-sequence model.

This approach will ease the problem of the AST size, because the length of a root-to-leaf path is shorter than the size of whole AST, unless the AST has no branching.

14.3 Improvement on Datasets

As is seen in Figure 9 and figure 10, most of the data are short length C codes. This is due to our data generation method, which enumerates

all of the subtree of a C source code. We intended the models to learn the structure of long data from the short data, though the result showed that the poorer decompilation results were generated from the longer data. Therefore, the length distribution of the training data might affect the training result. To cope with this problem, we will need to seek an effective distribution of the length for training.

参考文献

[1] Chainer. URL `https://chainer.org/`.

[2] Clang: a c language family frontend for llvm. URL `https://clang.llvm.org/`.

[3] Github. URL `https://github.com/`.

[4] pycparser. URL `https://github.com/eliben/pycparser`.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL `http://arxiv.org/abs/1409.0473`.

[6] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. 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)*, pages 353–368, 2013.

[7] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018. URL `http://arxiv.org/abs/1802.03691`.

[8] Distler Dennis. Malware analysis: An introduction. February 2008. URL `https://www.sans.org/reading-room/whitepapers/malicious/malware-analysis-introduction-2103`.

[9] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013. URL `http://arxiv.org/abs/1303.5778`.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[11] Guilfanov Ilfak. Decompilers and beyond, 2008. URL https://www.hex-rays.com/products/ida/support/ppt/decompilers_and_beyond_white_paper.pdf.

[12] AV-TEST Institute. URL https://www.av-test.org/en/statistics/malware/.

[13] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 20–30, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5714-2. doi: 10.1145/3196321.3196330. URL http://doi.acm.org/10.1145/3196321.3196330.

[14] Haglund Jonas. A case study of four recent high-impact malware attacks. 2017. URL https://www.engsec.se/wp-content/uploads/2018/06/Engvall-Security-Malware-Case-Studies-Jonas-Haglund.pdf.

[15] Eran Kalige, Darrell Burkey, and IPS Director. A case study of eurograbber: How 36 million euros was stolen via malware. *Versafe (White paper)*, 35, 2012.

[16] Neel Kant. Recent advances in neural program synthesis. *CoRR*, abs/1802.02353, 2018. URL http://arxiv.org/abs/1802.02353.

[17] D. S. Katz, J. Ruchti, and E. Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 00, pages 346–356, March 2018. doi: 10.1109/SANER.2018.8330222. URL doi.ieeecomputersociety.org/10.1109/SANER.2018.8330222.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

[19] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[20] Frankie Li, Anthony Lai, and Ddl Ddl. Evidence of advanced persistent threat: A case study of malware for political espionage. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 102–109. IEEE, 2011.

[21] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM. ISBN 1-58113-738-9. doi: 10.1145/948109.948149. URL `http://doi.acm.org/10.1145/948109.948149`.

[22] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015. URL `http://arxiv.org/abs/1508.04025`.

[23] Jerome Miecznikowski and Laurie Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *International Conference on Compiler Construction*, pages 111–127. Springer, 2002.

[24] Alexios Mylonas, Stelios Dritsas, Bill Tsoumas, and Dimitris Gritzalis. Smartphone security evaluation-the malware attack case. *SECRYPT*, 11:25–36, 2011.

[25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-jing Zhu. Bleu: a method for automatic evaluation of machine translation. 10 2002. doi: 10.3115/1073083.1073135.

[26] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In Yan Shoshitaishvili and Ruoyu (Fish) Wang, editors, *Workshop on Binary Analysis Research*, San Diego, CA, USA, February 18-21 2018. URL `http://www.cs.unm.edu/~eschulte/data/bed.pdf`.

[27] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, Nov 1997. ISSN 1053-587X. doi: 10.1109/78.650093.

[28] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL `http://arxiv.org/abs/1409.3215`.

[29] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10 pp.–54, Nov 2005. doi: 10.1109/WCRE.2005.13.

[30] Mike Van Emmerik and Trent Waddington. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 27–36, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2243-2. URL `http://dl.acm.org/citation.cfm?id=1038267.1039035`.

[31] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 683–693, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106289. URL `http://doi.acm.org/10.1145/3106237.3106289`.

[32] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696, 2017. URL `http://arxiv.org/abs/1704.01696`.