
An Empirical Assessment on the Limits of Binary Code Summarisation with Transformer-based Models

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ali Al-Kaswan

born in Lelystad, the Netherlands

To be defended publicly
on 21 July 2022



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS
Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl



DECAL Lab
Faculty of Computer Science
University of California Davis
Davis, the United States
www.cs.ucdavis.edu

An Empirical Assessment on the Limits of Binary Code Summarisation with Transformer-based Models

Author: Ali Al-Kaswan

Student id: 4679678

Abstract

Reverse engineering binaries is required to understand and analyse programs for which the source code is unavailable. Decompilers can transform the largely unreadable binaries into a more readable source code-like representation. However, many aspects of source code, such as variable names and comments, are lost during the compilation and decompilation processes. Furthermore, by stripping the binaries, more informative symbols/tokens, including the function names, are also removed from the binary.

Reverse engineering is time-consuming, much of which is taken up by labelling the functions with semantic information. Therefore, we propose a novel code summarisation method for decompiled and stripped decompiled code. First, we leverage the existing BinSwarm dataset and extend it with aligned source code summaries. Next, we create an artificial demi-stripped dataset by removing the identifiers from unstripped decompiled code. To train our model for summarising code using this dataset, we fine-tune a pre-trained CodeT5 model for the code summarisation task on the given dataset. Furthermore, we investigate the performance of the input types, the impact of data duplication and the importance of each aspect present in the source code on the model performance. Moreover, we design and present some intermediate-training objectives to increase the model performance.

We present the following findings: Firstly, we find that the model generates good summaries for decompiled code, with similar performance to source C code. Compared to summarising decompiled code, the quality of the

demi-stripped model is significantly lower but still usable. Stripped performed worse and produced mostly incorrect and unusable summaries. Secondly, we find that deduplication greatly reduces the performance of the model, putting the performance of decompiled code roughly in line with other decompiled datasets. Thirdly, we found that the loss of identifiers causes a drop in the BLEU-4 score of 35%, with another 25% decrease attributable to the increase of decompilation faults caused by stripping. Lastly, we show that our proposed deobfuscation intermediate-training objective improves the model's performance by 0.54 and 1.54 BLEU-4 on stripped and demi-stripped code, respectively.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Dr. S. Verwer, Faculty EEMCS, TU Delft
External Supervisor:	Prof. Dr. P. Devanbu, Faculty Computer Science, UC Davis
Daily Supervisor:	Dr. M. Izadi, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. A. Sawant, Faculty Computer Science, UC Davis

Preface

Dear reader,

The last nine months have been a challenging but incredible journey. I was not well acquainted with the subject of the thesis, so I was continuously pushed further and further away from my comfort zone as the project progressed. While this has been a very educational and rewarding journey, it would not have been possible without the help of many wonderful people.

First and foremost, I would like to thank Anand Ashok Sawant, who introduced me to this topic and whose insights were invaluable to the project. Many thanks to Toufique Ahmed for always being available to discuss difficulties and answering my many questions. Thanks to Prof. Arie van Deursen for introducing me to the topic and the right people, for the many valuable insights and for keeping the project on track. Thanks to Maliheh Izadi for sharing her extensive knowledge on the topic, for always being available for questions and her help writing the thesis. Many thanks to Prof. Prem Devanbu for always providing profound insights and his academic curiosity during the weekly meetings. Also, I would like to thank the DECAL lab members, and specifically Toufique, Prem and Anand, for their hospitality during my visit to Davis.

I would also like to thank my friends and colleagues at Eye Security for their continued support and the Friday evening hacking sessions, which were a welcome distraction. Finally, I thank my friends and family for always being there for me and supporting me throughout this journey.

Ali Al-Kaswan
Delft, the Netherlands
July 14, 2022

Contents

Preface	iii
Contents	v
List of Figures	vii
1 Introduction	1
1.1 Problem Definition and Significance	1
1.2 Contributions	2
1.3 Implications	3
1.4 Structure	4
2 Background	5
2.1 Binary Reverse Engineering	5
2.2 Natural Language Processing for Code	10
3 Related Work	19
3.1 Recovering Identifiers from Stripped Binaries	19
3.2 Binary Translation	20
3.3 Automatic Source Code Summarisation	20
4 Approach	23
4.1 Data Collection	23
4.2 Intermediate-Training	26
4.3 Fine-Tuning	27
5 Experimental Setup	29
5.1 Research Questions	29
5.2 Dataset	30

CONTENTS

5.3	Model Configuration	32
5.4	Manual Evaluation	34
5.5	Intermediate-Training	34
6	Results	37
6.1	RQ1: Data-richness study	37
6.2	RQ2: Data-duplication study	38
6.3	RQ3: Data-input study	38
6.4	RQ4: Model-objective study	39
7	Discussion	41
7.1	Interpreting the Results	41
7.2	Threats to validity	43
7.3	Broader Impacts	45
7.4	Ethical Considerations	46
8	Conclusion and Future Work	49
8.1	Summary	49
8.2	Future Work	49
	Bibliography	53
A	Glossary	59
B	Extreme Summarisation	61
B.1	Methodology and Experimental Setup	61
B.2	Results	62

List of Figures

1.1	Proposed solution	3
2.1	secp256k1	7
2.2	Transformation of binary to readable code	7
2.3	Ghidra’s disassembly window.	8
2.4	Ghidra’s decompiler window showing a snippet of decompiled function from the secp256k1 ECDSA library	9
2.5	Model overview of Transformer [36], with an encoder (left) and de- coder (right).	11
4.1	Data Collection	24
4.2	Pre-Training	26
5.1	Example function with its documentation (truncated)	30
5.2	Example entries of a .jsonl file with two dummy entries	32
5.3	Translation intermediate training objective	35
5.4	Deofbuscation intermediate training objective	35
5.5	Span-detection intermediate training objective	36
7.1	Short decompiled function from the GNU Debugger [14] decompiled by Ghidra	43
7.2	Example of an offensive comment from radareorg/radare2-bindings/radare2 /libr/anal/esil.c,	46
B.1	Loss during CodeBERT training on stripped data	62

Chapter 1

Introduction

1.1 Problem Definition and Significance

Reverse Engineering (RE) is the act of “breaking down” the inner workings of binary executables to analyse and understand programs. It is used to find vulnerabilities and analyse novel malware. Reverse engineering can also help researchers quickly understand novel malware, fingerprint existing malware [37], replicate software of which the source code is lost, discover illegitimate usages of intellectual property, porting abandonware, and more [10]. The binaries are the most accurate representation of the program that runs on the system, there is no guarantee that the source code represents the binary delivered to the user [10].

Unlike binaries, source code is relatively easy to read, but unfortunately, it is not always available. The original source code is compiled into a runnable binary program by compilers such as Clang/LLVM¹ or GCC² and delivered to the user. Once compiled, the exact source code version, which was used for compilation, may become unknown or lost altogether.

To understand what a binary program does exactly, the binary code can be decompiled into readable code by decompilers such as Ghidra³ and IDA Pro.⁴ Understanding decompiled code is still an intrinsically difficult process. It is a manual, time-consuming process and still largely depends on the skill of the Reverse Engineer [10, 37]. Much of the work that goes into reverse engineering a binary is spent labelling functions with semantic information [37].

Source code can be described as having two information channels: the algorithmic channel and the natural language channel. The algorithmic channel specifies the execution of a program (semantics), while the natural language channel specifies

¹Clang: <https://clang.llvm.org/>

²GCC: <https://gcc.gnu.org/>

³Ghidra Framework: <https://ghidra-sre.org/>

⁴IDA Pro: <https://hex-rays.com/ida-pro/>

the purpose and context of the program (syntax) [11]. The natural channel includes function and variable names, code comments and the specific human-readable structure of programs. Computers only consider the algorithmic channel to execute a program, while humans use both the algorithmic channel and the natural channel to understand a piece of code [11]. Furthermore, code is very regular and predictable, even more so than natural languages [20]. Language models leverage this naturalness of code [20] to understand source code.

Due to the naturalness of code, Natural Language Processing (NLP)-based techniques are tailored to the source code as well. For instance, code summarization [25] is used to automatically generate short natural language descriptions of code. While these methods have been successfully applied to programming languages such as Python, Java and PHP [38, 17, 12, 2], none of these methods has been applied to the relatively syntactically-poor output of decompilers. The compilation process, which transforms readable code into runnable binaries, destroys much of the information contained in the natural channel. Especially stripped binaries - binaries of which the symbol table is removed - will be challenging since they have almost no identifiers at all. Hence, in this thesis we aim to investigate the following goal to address this gap:

To investigate the application of state-of-the-art code summarisation methods for decompiled code.

1.2 Contributions

We, therefore, propose our code summarisation model, which takes decompiled functions and synthesises summaries. Figure 1.1 shows an overview of the proposed approach. The starting point for a reverse engineer is a binary that a compiler has compiled from source code. This binary is then processed by a reverse engineering tool like Ghidra, which decompiles the binary. From this decompiled code, the functions are then extracted. These decompiled functions are then summarised using our trained CodeT5 [38] model, which will be discussed in more detail in Chapter 2.

We perform experiments on this solution to find the impact of decompilation and stripping on the model performance. Furthermore, we investigate the effect of duplicates in the data, and which differences between stripped and unstripped data contribute most to the performance drop. Finally, we use the knowledge from these experiments and design intermediate-training objectives to improve the model's performance on stripped decompiled code.

Our main contributions can be summarised as follows:

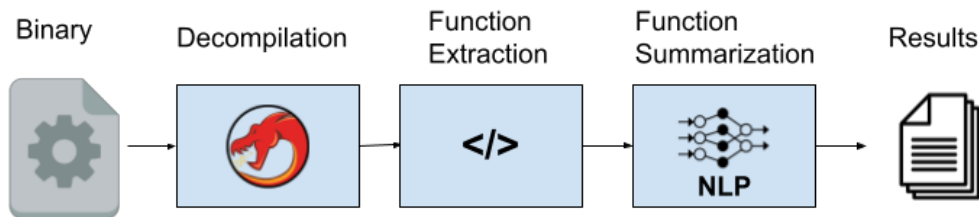


Figure 1.1: Proposed solution

Model: Our main contribution is a pre-trained, fine-tuned code-summarisation model for decompiled and stripped-decompiled code. The model uses CodeT5 and is fine-tuned and evaluated on our own dataset of source code, decompiled code and stripped-decompiled code pairs.

Impact study: To create this model, we explored the influence of the input types, the impact of data duplication, and the impact of different aspects of stripped and unstripped decompiled code on the model performance.

Intermediate-training: To improve the performance of the model, we implemented and evaluated the Neural Machine Translation (NMT), deobfuscation (DOBF) and span-detection (SPAN) intermediate-training objectives.

Dataset: Finally, we contribute the dataset used to fine-tune and pre-train the model. The dataset contains aligned comment-source code, comment-decompiled code, and comment-stripped-decompiled code pairs. We also provide a synthetic dataset of comment-demi-stripped code pairs. Furthermore, we provide the data used for the pre-training objectives. This novel dataset can be used by other works to train and evaluate their models.⁵

1.3 Implications

Several stakeholders can benefit from this research:

1. Firstly, security researchers who aim to understand malware. This could help them understand and reverse engineer novel malware more quickly.
2. Users of closed-source software can use this to inspect the software for faults. Closed source software can be patched, rewritten and reused to serve its exact purposes. Furthermore, understanding the source code can allow users to change the binary in memory during runtime. A malicious example of this

⁵Dataset: <https://doi.org/10.4121/20301309>

group is game hackers who change certain memory addresses to give themselves an unfair advantage (changing their health points, for instance) [10].

3. Reverse engineers aim to replicate closed-source software or software of which the source code is lost. Reverse engineering can be used to help create open-source copies or to port the software to newer architectures. Examples range from porting old abandoned open-source projects to porting abandonware (on which copyrights still apply), or even the theft of intellectual property from closed-source commercial software.
4. On the flip side, creators of closed-source software can use reverse engineering to determine whether other software has copied their products and infringed on their intellectual property.
5. Lastly, developers of reverse engineering programs and toolkits might be able to use the results of this research to enhance their own products.

1.4 Structure

The rest of the thesis is outlined as follows: Chapter 2 will consist of a brief introduction and explanation of concepts and tools used throughout the thesis. In Chapter 3 we will discuss other relevant work in this field and how these relate to ours. Chapter 4 will cover the research methodology, the experimental setup will be covered in Chapter 5. The results will be presented in Chapter 6, followed by a discussion on the findings and a short discussion on the threats to validity and ethical considerations in Chapter 7. Finally, Chapter 8 will conclude with the findings and discuss possible future works.

Chapter 2

Background

In this chapter, we will cover the required background knowledge to understand this thesis and discuss the corresponding works. We will start by covering different aspects of binary reverse engineering. Firstly, compilers and compiler optimization levels will be covered, and then stripping will be explained. Finally, we will discuss the Ghidra toolkit and the relevant modules. The second part of this chapter will feature an explanation of Natural Language Processing methods for code. We will start by explaining the code summarisation task, and we will then move on to explain transformers and CodeT5. Finally, the BLEU, METEOR and ROUGE evaluation metrics will be explained in detail.

2.1 Binary Reverse Engineering

2.1.1 Compilers and Optimization Levels

Compilers are programs that translate code from one language to another, but generally, and in the context of this thesis, the term is used to refer to programs that translate high-level code, like C, to a lower-level language such as machine code. For our work we focus on the GNU Compiler Collection (GCC)¹ and Clang/LLVM (Clang).²

Compilers frequently feature optimization levels. Generally, the goal of optimizations is the improvement of runtime performance or program size at the expense of compilation time and the ability to debug [21]. Compilers generally use optimization flags, grouped into optimization levels, where each level uses a different set of optimization flags.

For example, the GCC features 60 optimization flags across 8 different optimization levels, which are denoted by a `-O` option [21, 22]. By default, if GCC

¹gcc: <https://gcc.gnu.org/>

²Clang: <https://clang.llvm.org/>

2. BACKGROUND

is invoked without any optimization options, the program will be compiled with O0. O1, O2 and O3 incrementally apply more optimization to the binary at the expense of a higher compilation time [22]. These optimization levels are also found in other compilers such as Clang-LLVM. Other optimization levels, such as Os, which optimises for binary size, are also included in GCC [22].

Optimizations can restructure and transform the program in relation to the source code, by changing the control flow or the data of the program [8]. This obfuscation can complicate the reverse engineering process by reducing the accuracy of Ghidra [8].

2.1.2 Stripping

Aside from compiling with higher optimization levels, binaries can also be stripped to obfuscate the underlying code and to resist analysis[39]. Binaries which have not been stripped still contain a lot of debugging information, which can be used during development. This debug information, like function names, self-defined types etc., can be used to analyse and reverse engineer the binary. Commercial Off-the-Shelf (COTS) software is often stripped to reduce the memory and storage footprint of binary, and to resist analysis to protect the intellectual property of the creator. Many vulnerable and malicious binaries are, unfortunately, also stripped to resist security analysis and hide their faults[19].

Unix and Unix-like operating systems include a strip utility. The strip utility removes any operands that are not necessary for the execution of the binary while ensuring that the execution of the binary remains unchanged. The exact implementation and scope of the utility is left to the implementation.³

The strip utility as implemented in GNU/Linux removes the symbol table from the binary. The symbol table contains each symbols location, type and name. The symbol table can be dumped for a given binary by using the nm command 2.1, which is included in Unix and Unix-like operating systems.⁴

Like higher optimization levels, the use of stripping can greatly complicate the efforts to reverse engineer a binary, as well as reduce the accuracy and effectiveness of reverse engineering tools.

³strip: <https://pubs.opengroup.org/onlinepubs/007908799/xcu/strip.html>

⁴nm: <https://pubs.opengroup.org/onlinepubs/9699919799/utilities/nm.html>

⁵Bitcoin secp256k1: <https://github.com/bitcoin-core/secp256k1>


```

U malloc
U memcpy
U memset
0000000000000060 r minus_b1.3052
0000000000000040 r minus_b2.3053
00000000000005680 t nonce_function_rfc6979
00000000000000c1 r one.4283
0000000000000120 r output32.4535
00000000000000e0 r pad.4239
00000000001101e0 r secp256k1_const_lambda
0000000000110240 r secp256k1_const_modinfo_fe
0000000000110200 r secp256k1_const_modinfo_scalar
000000000000c7d0 T secp256k1_context_clone
000000000000c690 T secp256k1_context_create
000000000000c920 T secp256k1_context_destroy
0000000000000000 D secp256k1_context_no_precomp
...

```

Figure 2.1: Sample output of nm command from secp256k1⁵ECDSA library

2.1.3 Ghidra

Ghidra is a free and open-source reverse engineering toolkit developed by the US National Security Agency. Ghidra has been in development since the turn of the century and had been in use internally, before being open-sourced in April of 2019.

Ghidra contains many separate analysis modules that allow a reverse engineer to analyse compiled code. The modularity of Ghidra and the inclusion of a scripting engine allow users to add custom modules and scripts. We will specifically focus on the tools used in the process that was used to transform binaries into readable code (see 2.2).

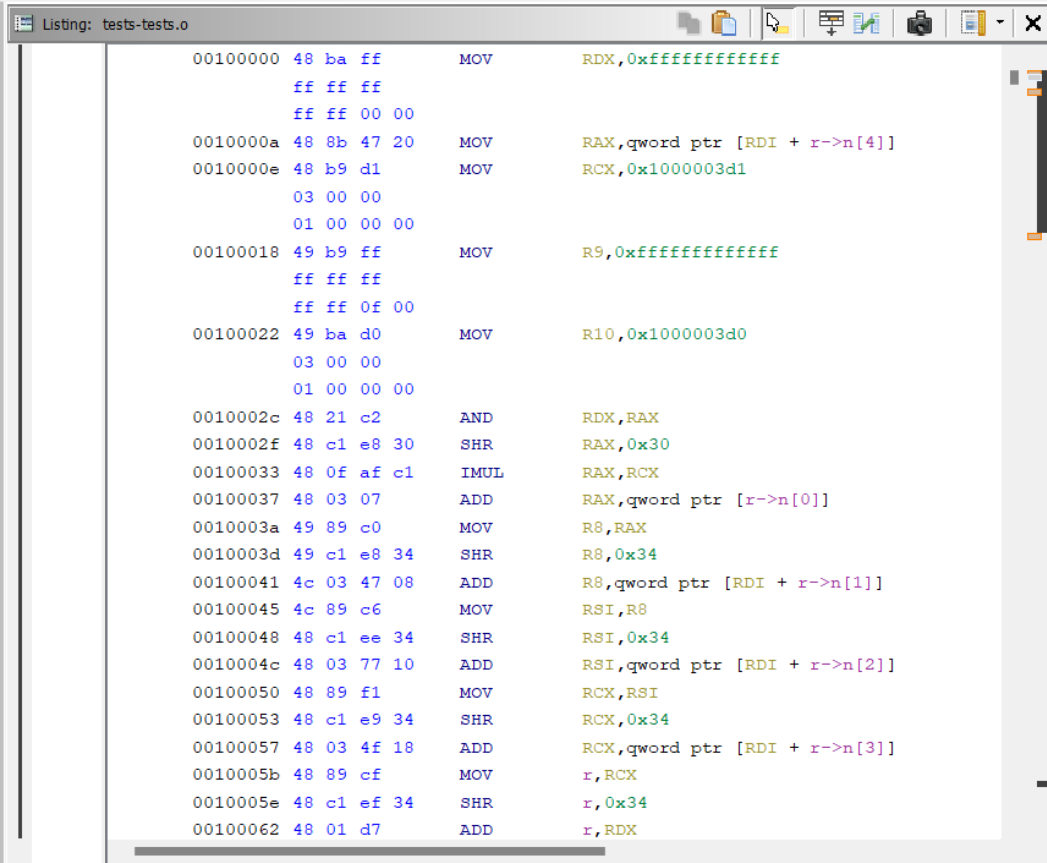


Figure 2.2: Transformation of binary to readable code

Ghidra features a disassembler, as seen in figure 2.3, which will take the binaries

2. BACKGROUND

and assemble them back into an intermediate representation. In the case of x86-x64 binaries like the binaries this project will focus on, this intermediate representation will be the assembly language. Processors have an associated language that defines the mapping between user-readable assembly language instructions (e.g. MOV, ADD, etc.) and their corresponding byte values. In order to properly disassemble a binary image for a specific architecture, Ghidra requires a language module for that specific processor. A language module is software that implements language translation. Ghidra has a set of language modules for the most popular processor languages (such as x86-64, ARM, MIPS ..). Besides these architectures which are supported out-of-the-box, developers have also been extending Ghidra's support with custom processor architectures. For example, there exists a language module for the Allegrex CPU featured in the PlayStation Portable.⁶



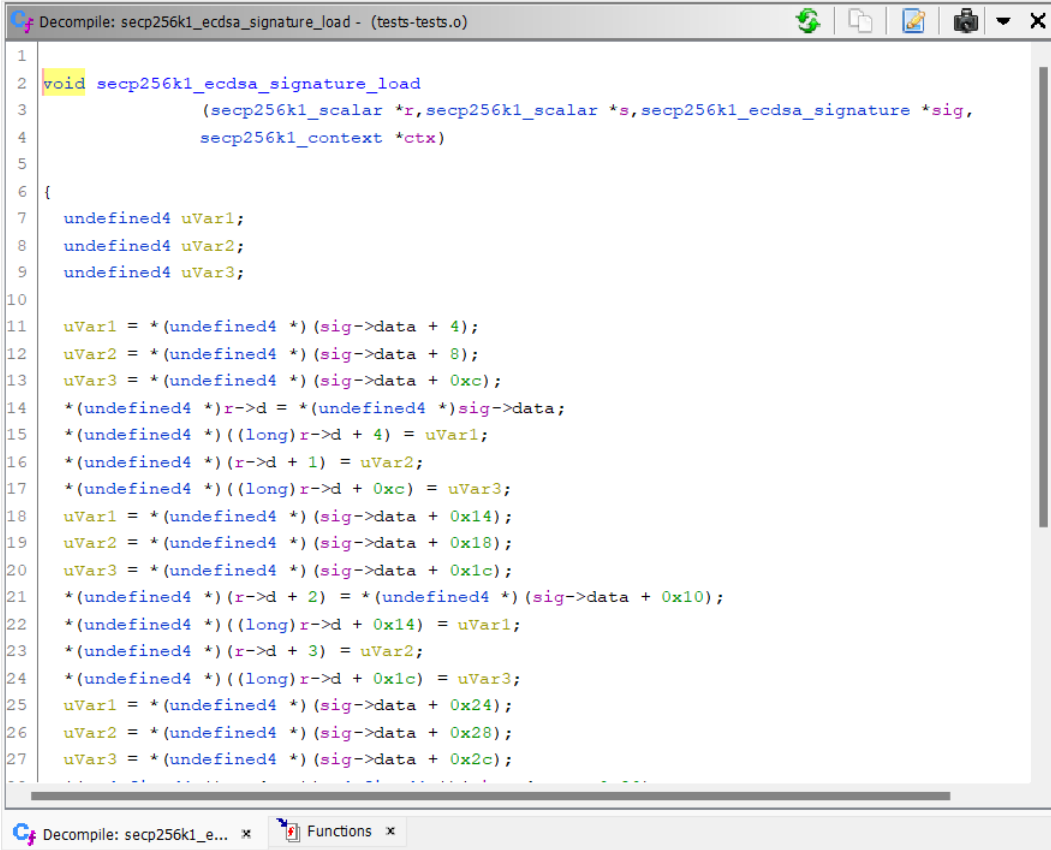
```
Listing: tests-tests.o
00100000 48 ba ff      MOV     RDX,0xffffffffffff
          ff ff ff
          ff ff 00 00
0010000a 48 8b 47 20   MOV     RAX,qword ptr [RDI + r->n[4]]
0010000e 48 b9 d1     MOV     RCX,0x1000003d1
          03 00 00
          01 00 00 00
00100018 49 b9 ff     MOV     R9,0xffffffffffff
          ff ff ff
          ff ff 0f 00
00100022 49 ba d0     MOV     R10,0x1000003d0
          03 00 00
          01 00 00 00
0010002c 48 21 c2     AND     RDX,RAX
0010002f 48 c1 e8 30   SHR     RAX,0x30
00100033 48 0f af c1   IMUL   RAX,RCX
00100037 48 03 07     ADD     RAX,qword ptr [r->n[0]]
0010003a 49 89 c0     MOV     R8,RAX
0010003d 49 c1 e8 34   SHR     R8,0x34
00100041 4c 03 47 08   ADD     R8,qword ptr [RDI + r->n[1]]
00100045 4c 89 c6     MOV     RSI,R8
00100048 48 c1 ee 34   SHR     RSI,0x34
0010004c 48 03 77 10   ADD     RSI,qword ptr [RDI + r->n[2]]
00100050 48 89 f1     MOV     RCX,RSI
00100053 48 c1 e9 34   SHR     RCX,0x34
00100057 48 03 4f 18   ADD     RCX,qword ptr [RDI + r->n[3]]
0010005b 48 89 cf     MOV     r,RCX
0010005e 48 c1 ef 34   SHR     r,0x34
00100062 48 01 d7     ADD     r,RDX
```

Figure 2.3: Ghidra's disassembly window.

The decompiler, also featured in Ghidra, is a processor language-agnostic trans-

⁶Ghidra-Allegrex: <https://github.com/kotcrab/ghidra-allegrex>

formation engine that takes the disassembled code and creates a source code representation, as can be seen in Figure 2.4. The representation is written in pseudo-C, a C-like representation that generally follows the general language conventions of C. The interactive decompiler window allows the user to directly see the correspondence between the pseudo-C and assembly representations. The user can also make changes to the decompiled code, such as changing automatically generated variable names and types and adding comments.



```
1
2 void secp256k1_ecdsa_signature_load
3     (secp256k1_scalar *r, secp256k1_scalar *s, secp256k1_ecdsa_signature *sig,
4     secp256k1_context *ctx)
5
6 {
7     undefined4 uVar1;
8     undefined4 uVar2;
9     undefined4 uVar3;
10
11     uVar1 = *(undefined4 *) (sig->data + 4);
12     uVar2 = *(undefined4 *) (sig->data + 8);
13     uVar3 = *(undefined4 *) (sig->data + 0xc);
14     *(undefined4 *) r->d = *(undefined4 *) sig->data;
15     *(undefined4 *) ((long)r->d + 4) = uVar1;
16     *(undefined4 *) (r->d + 1) = uVar2;
17     *(undefined4 *) ((long)r->d + 0xc) = uVar3;
18     uVar1 = *(undefined4 *) (sig->data + 0x14);
19     uVar2 = *(undefined4 *) (sig->data + 0x18);
20     uVar3 = *(undefined4 *) (sig->data + 0x1c);
21     *(undefined4 *) (r->d + 2) = *(undefined4 *) (sig->data + 0x10);
22     *(undefined4 *) ((long)r->d + 0x14) = uVar1;
23     *(undefined4 *) (r->d + 3) = uVar2;
24     *(undefined4 *) ((long)r->d + 0x1c) = uVar3;
25     uVar1 = *(undefined4 *) (sig->data + 0x24);
26     uVar2 = *(undefined4 *) (sig->data + 0x28);
27     uVar3 = *(undefined4 *) (sig->data + 0x2c);
--
```

Figure 2.4: Ghidra’s decompiler window showing a snippet of decompiled function from the secp256k1 ECDSA library

2.2 Natural Language Processing for Code

2.2.1 Code Summarisation

Code summarisation (also referred to as source code summarisation) is the task of writing short descriptions from source code, usually a single sentence summary of the source code. The main use is for software documentation, like the one-sentence JavaDoc description used in Java [25]. This documentation is important for program comprehension and for maintenance. But the process of writing and maintaining these descriptions is a labour-intensive and time-consuming task, which is where the need to automate that process arises. Code summarisation is an extremely active and popular research problem in the field of software engineering [25].

While code summarisation can be applied to any particular piece of code, the problem is usually posed as the generation of a description on the function or method level. A major limitation of these methods is that the model is not provided with any background knowledge.

A twist on the code summarisation task is proposed by Allamanis et al.. Extreme summarisation is defined similarly to the code summarisation task, but instead of an entire sentence, the model is tasked with summarizing the function in the form of a single descriptive function name.

2.2.2 Transformers and CodeT5

In this thesis, the CodeT5 model [38], which is based on Transformers [36], is extensively used. This section provides a quick overview of the Transformer architecture and the CodeT5 model.

Transformers

The current state-of-the-art NLP models for programming languages such as CodeT5 [38], CodeBERT [17] and Codex [12] are all based on the Transformer architecture [36].

Transformers were originally proposed by Vaswani et al. as a novel sequence-to-sequence [35] (seq2seq) architecture. Unlike the Recurrent Neural Networks [32] (RNN), the Long Short-Term Memory [33] (LSTM) variant of RNNs [32] and Convolutional Neural Networks [26] (CNN), Transformers only use a mechanism called self-attention to capture dependencies between the input and output.

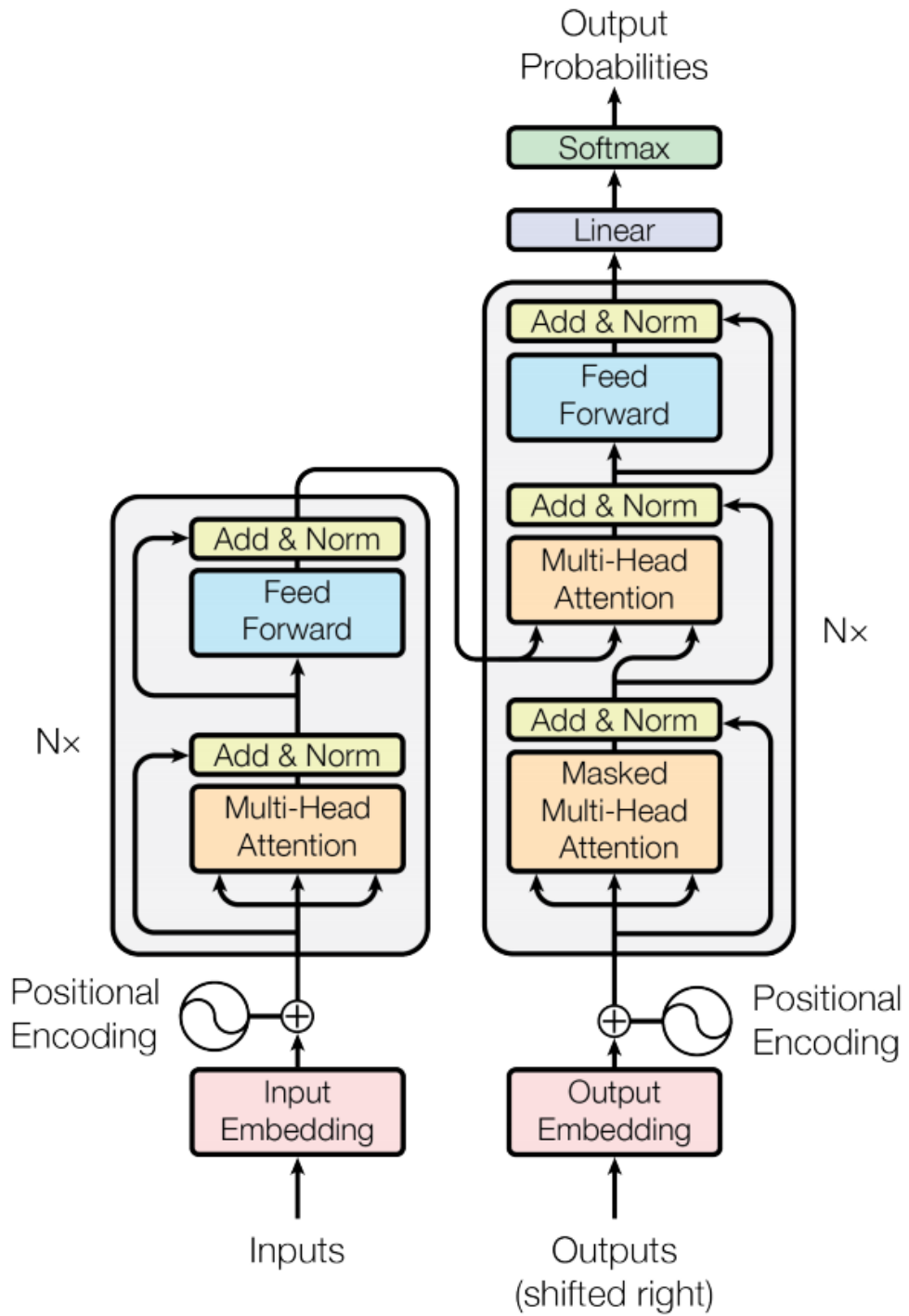


Figure 2.5: Model overview of Transformer [36], with an encoder (left) and decoder (right).

2. BACKGROUND

Transformers use an encoder-decoder architecture 2.5. The encoder maps a sequence of input representations $x = (x_1, x_2 \dots x_n)$ to a continuous intermediate representation $z = (z_1, z_2 \dots z_n)$. From this intermediate representation the decoder creates an output $y = (y_1, y_2 \dots y_m)$.

The encoder is comprised of N layers. The input tokens are transformed into an embedding, to every embedded token a positional encoding is added. This positional encoding adds information about the position of every token in the sequence. RNNs receive an $O(n)$ time penalty for sequential operations on sequences of length n , while sequential operations in Transformers are $O(1)$ in respect to the length n of the sequence.

The input is then processed by the self-attention function, which maps a query, and a set of key-value pairs to an output $Attention(Q, (K, V))$. Intuitively, the query vector Q can be described as the input token, and the mapping finds the most similar key vector K and outputs the value. This is done by taking the dot product of Q and K , normalizing by dividing by the dimension K and taking the softmax function. This normalization is important to limit the dot-product values to ensure that the softmax function still retains a good gradient flow. The resulting value $softmax(\frac{QK^T}{\sqrt{d_k}})$ is maximal when the vectors Q and K are most similar. Finally this can be multiplied with the value V to get the value paired with the most similar value.

$$Attention(Q, (K, V)) = softmax(\frac{QK^T}{\sqrt{d_k}}V)$$

This attention function is applied h times and the resulting concatenation is called the Multi-Head attention layer, where in every attention head different learned weight matrices W_i are also applied.

$$MultiHead(Q, K, V) = Concat(head_1, head_2 \dots head_h)W^O$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

This output is then fed into a Feed-Forward Neural Network and normalized. This single encoder layer, can be stacked N times. This entire process can be parallelized to pass the entire input sequence at once through the encoder.

The decoder is similarly structured to the encoder. Firstly the target sequence is shifted to the right, such that the model has to predict the next target token. The target sequence is also embedded and the positional encoding is added. The first Multi-Head attention block has masking applied, such that the attention of tokens further in the target sequence is not taken into account while predicting the attention of the current token. Recall that, during training, the model can use the entire input sequence, but only the previous target sequences to predict a token. The mask is a simple upper-triangular matrix, where the upper triangle is set to $-\infty$ and the lower

triangle to 0, such that:

$$\text{Attention}(Q, (K, V)) = \text{softmax}(\text{mask} + \frac{QK^T}{\sqrt{d_k}}V)$$

This ensures that, during training, the self-attention blocks can still be computed in parallel, without having to resort to sequentially sampling the target. The next attention block in the decoder implements cross-attention, where the key and value K, V vectors are fed in from the encoder, and the query Q vectors are fed in from the previous decoder block. This allows every token in the decoder to attend to all tokens in the encoder, which is important to allow the output to condition to the input. Similarly to the encoder, the final block of the decoder is a feed-forward neural network. The decoder is also stacked N times, and the output of these layers is fed into a Linear Feed Forward Neural Network, which transforms the embedding into a vector of probabilities for every token in the vocabulary of the model. Then the softmax function is applied to the output of the network and the highest probability token is selected as the output. During training, the model is tuned to maximize the probability of predicting the correct output token given the input to the encoder.

Transformers offer a variety of advantages over RNNs and CNNs:

Parallelizability Transformers can process the entire input sequence in parallel.

This allows them to leverage the high number of processing cores in conventional GPU architectures, which makes them much faster to train and deploy.

Long Memory The Self-Attention paradigm allows the model to establish and remember dependencies along the entire length of the input sequence. The length of the input does not restrict the dependencies. Furthermore, because the model is not recursive in nature and the self-attention vector is calculated for the entire input, the memory gradient does not vanish as with CNNs.

Interpretability Unlike the black-box-nature of RNNs and CNNs the self-attention gradient of the multi-head attention layers can be visualized to show the dependencies between different tokens of the input and output.

Transfer Learning

Pre-trained Transformers-based language models, such as RoBERTa [29], CodeBERT [17] and CodeT5 [38] utilize a pre-train and fine-tune paradigm. In this paradigm, the models are first trained in an unsupervised manner on a large unlabeled dataset. In the case of RoBERTa, a Masked Language Modeling (MLM) objective was used, where random tokens are masked out from a swath of text (or code in CodeBERT's case) and the model is tasked to predict said tokens. These pre-trained models can then be fine-tuned to perform a more specialized task, such

2. BACKGROUND

as summarisation. This concept is called transfer learning (TL). TL uses the knowledge that is obtained in one task to solve a different task. It allows the creation of general models that are trained once on massive (usually unlabeled) datasets. These general models, which contain general domain knowledge can then be trained for a specific downstream task. In general, this approach is quicker and requires less training data than training a model on the downstream task from scratch [15].

CodeT5

CodeT5 [38] is another state-of-the-art pre-trained programming language model built on the T5 (Text-to-text Transfer Transformer) architecture proposed by Wang et al. and pre-trained on a mix of unsupervised and supervised tasks. CodeT5 employs an encoder-decoder architecture. Some state-of-the-art models, such as BERT [15] only feature an encoder, while other models, such as GPT [9] only feature a decoder. Furthermore, in contrast to other models, CodeT5 is trained using both unimodal (PL only) and bimodal (NL-to-PL) tasks in six programming languages. This bimodal training allows CodeT5 to have strong performance in both uni-modal (PL-to-PL) tasks such as code translation and code refinement, as well as cross-modal tasks such as code summarisation and code generation (PL-to-NL).

2.2.3 Scoring Methods

To evaluate the quality of the source code summaries an evaluation metric is required, which quantifies the quality of a produced candidate summary compared to a reference. The simplest metric is an exact match score, where the score is 1 if the candidate matches the reference exactly. This metric does not reflect the actual quality of the produced summary.

BLEU

The most widely used metric in the code summarisation task is Bilingual Evaluation Understudy Score (BLEU) [34]. BLEU was originally proposed as a technique to evaluate the machine translations against a set of human translations. In the code-summarisation domain, the automatically generated summaries are compared against a programmer defined summary. BLEU produces a percentage number between 0 and 100, which defines the similarity between a candidate and a set of reference sentences. BLEU-N calculates the cumulative N-gram precision scores, the number of matching N-grams divided by the total number of N-grams in the candidate sentence. The unigrams (1-grams) account for the adequacy of the candidate while the longer N-grams account for the fluency. Two problems arise, firstly the score can be artificially inflated by simply repeating the same matching N-gram

multiple times in the candidate 2.1. Secondly, the score can also be inflated by creating a very short candidate such that the denominator is very small 2.1.

Reference	Calculate	the	Secp256k1	private	key	given	generator	Precision
Repeating	the	the	the	the	the	the		6/6 = 1
Short	Calculate							1/1 = 1

Table 2.1: An example of a repeating and short candidate, and the resulting 1-gram precision.

To counteract these two issues, BLEU firstly clips the n-grams in the numerator, such that each match is only counted once. Secondly, a Brevity Penalty (BP) is applied such that a short candidate is penalized. Recall that long candidates are already inherently penalized by the precision metric. Let c be the length of the candidate and r be the length of the reference, BP is then defined as:

$$BP = \begin{cases} 1, & \text{if } c < r. \\ e^{(1-r/c)}, & \text{otherwise.} \end{cases} \quad (2.1)$$

The geometric mean of each of the clipped n-gram precision scores p_n is taken:

$$\text{BLEU-N} = BP \cdot \exp\left(\sum_{n=1}^N w_n \log p_n\right)$$

Where w_n is a uniform weight $w_n = 1/N$.

While N could be any number, BLEU-4 is the most commonly used variant. Since code summaries are generally quite short and the BLEU score was originally designed for use in longer corpora of text it can occur that there are no matching 4-grams between a candidate and reference summary[34]. The geometric mean will then result in 0, and the BLEU score will also be 0. To solve this, smoothing is applied. There exist a variety of smoothing methods, which all add a factor to the numerator and denominator of p_n [34].

A major weakness of the BLEU metric is the fact that it does not consider the meaning of the reference and candidate texts, and that synonyms and syntactically different texts have a low score despite their adequacy. For example, a reference of “Calculate the Secp256k1 private key given the generator” and a candidate of “Secp256k1 secret key calculation from G” will score very lowly, despite the fact that they both convey the same information.

METEOR

METEOR (Metric for Evaluation for Translation with Explicit Ordering) [24] is also a metric proposed to assess a machine-generated translation against multiple human-generated references. Unlike BLEU which is a precision-based metric,

2. BACKGROUND

BLEU Score	Interpretation
<10	Almost useless
10 - 19	Hard to get the gist
20 - 29	The gist is clear, but has significant grammatical errors
30 - 40	Understandable to good translations
40 - 50	High quality translations
50 - 60	Very high quality, adequate, and fluent translations
>60	Quality often better than human

Table 2.2: Interpretation of BLEU scores [34]

METEOR is more recall-focused. METEOR has a higher correlation with human judgement than BLEU [25]. METEOR uses wordlists and stemming to also take synonyms into account. On the flip side, the stemming and synonyms are language-dependent and somewhat more expensive than BLEU to calculate.

The METEOR score is calculated by first generating an alignment between a candidate and the reference sentence. An alignment is a mapping between each unigram (or word) in the candidate and reference sentences. The unigrams can be matched in 3 ways: Firstly the unigrams can be an *exact* match. The unigrams can also be matched on the *stem*, which will match words on their stem. For example the words 'improved' and 'improving' will both be reduced to their stem 'improv' and will then be matched. Finally, words can also be matched if they are synonyms, for instance 'quick' and 'fast'. This is determined using the WordNet [16] database. The alignment which best preserves the word order is selected.

A fragmentation penalty (p) is also introduced to account for congruity. The minimum number of adjacent matched groups (or chunks) of words in the candidate and reference. The penalty Pen is calculated as a function of the number of chunks ch , the number of matches m , and hyperparameters γ, β .

$$frag = \frac{ch}{m} \quad (2.2)$$

$$Pen = \gamma * frag^\beta$$

The selected alignment is then scored using a harmonic mean:

$$P = \frac{m}{t}$$

$$R = \frac{m}{r}$$

$$F_{mean} = \frac{P * R}{\alpha * P * (1 - \alpha) * R} \quad (2.3)$$

$$Score = (1 - Pen) * F_{mean}$$

ROUGE-L

ROUGE or Recall-Oriented Understudy for Gisting Evaluation, is a package which includes a number of metrics, the most popular among them is ROUGE-L [28]. As the name implies, it is more recall oriented than BLEU. Unlike BLEU and METEOR, the ROUGE measures were constructed to score automatically generated candidate summaries against one or multiple reference summaries. ROUGE-L is much quicker than METEOR to calculate and is language-independent.

ROUGE-L simply finds the longest common subsequence (LCS) between the reference and the candidate. Note that the words do not need to be consecutive but they have to be in order. The length of the LCS lcs , the length of the reference r , the length of the candidate c , and a large constant $\beta \rightarrow \infty$ are then used to calculate the score:

$$\begin{aligned} P &= \frac{lcs}{c} \\ R &= \frac{lcs}{r} \\ ROUGE-L &= \frac{(1 + \beta^2) * R * P}{R + \beta^2 * P} \end{aligned} \tag{2.4}$$

Chapter 3

Related Work

In this section, we provide an overview of studies most related to our goal of generating summaries for binary functions. Binary reverse engineering and the use of NLP for software engineering are vast and active fields, so we select and discuss the closest state-of-the-art works in the field. We categorise the studies into three groups: identifier recovery, binary translation and code summarisation. Finally, we will discuss the open challenges and the relation of our own work to these challenges.

3.1 Recovering Identifiers from Stripped Binaries

Debin [19] aims to recover debug information from stripped binaries. The authors use a tree-based classification model to determine which registers in the intermediate representation are useful to predict and map to an actual variable, as opposed to constant values and intermediate values, which have no mapping to source code variables. A probabilistic graph-based model is built, and all the variable names and types are jointly recovered using a maximum a posteriori probability inference. The authors show state-of-the-art performance in the recovery of both names and types. Notably, the authors claim no noticeable performance penalty between different optimisation levels.¹

Unlike previous works, **VarBERT** [7] uses a Transformer-based NLP model for the task of variable name recovery. The authors pre-trained a BERT model with a standard Masked Language Modelling (MLM) objective. This same model is then fine-tuned to predict the names and types from unstripped binaries using a constrained Masked Language Modelling objective. The authors show state-of-the-art performance in the recovery of names and types. Furthermore, the authors of

¹Debin Presentation at CCS' 18: https://youtu.be/x1x_KtS-5Hs?t=1551

VarBERT have shown that the performance of Debin does not generalise well to other datasets [7].

FUNCRE [3] uses a pre-trained and fine-tuned ROBERTA model to predict usages of inlined library functions. Recall that compilers with optimisations enabled can inline functions in the binary (Chapter 2). Therefore, identifying library function calls is key to understanding the behaviour of the binary. The authors use indelible markers, which do not get destroyed by compiler optimisations, to mark usages of library functions. The model is pre-trained with an MLM objective on the stripped decompiled code. It is then fine-tuned to detect and name inlined function usages in a given context window. The authors combine their model with the function recovery functionality included in Ghidra. The resulting combined model significantly improved the performance of Ghidra’s inlined function recovery capabilities. The performance exceeded Hex-Ray’s IDA-pro, which performs significantly better than Ghidra at inlined function recovery [3].

3.2 Binary Translation

Neutron [27] frames decompilation as a neural machine translation problem and utilises an LSTM-based neural translation network to translate disassembled binaries back to C source code. The binaries are not stripped and do not have any optimisations enabled. To handle long-term information, the LSTM also includes an attention mechanism. The authors first remove the identifiers before translation, ensuring the identifiers are aligned with the translated identifier-less source code. The translations created by Neutron can contain syntax errors, so the authors apply regular expressions to create a tailor-made syntax checker. Neutron achieves high accuracy on the translation task, but only on unstripped and non-optimised code.

3.3 Automatic Source Code Summarisation

PolyglotCodeBERT [2] applies multilingual training to a CodeBERT [17] model. The use of multilingual and multimodal pre-training objectives is a well-established paradigm; recall the use of multilingual and bimodal pre-training objectives in CodeT5 [38] (Chapter 2). Ahmed and Devanbu propose a multilingual fine-tuning objective to augment the labelled fine-tuning dataset, to improve the performance of languages with less abundant labelled data. The authors found that identifiers are an essential aspect of training data and that identifiers are not language-specific. Recall that unstripped decompiled binaries have few identifiers, and that stripped decompiled binaries have no identifiers. We will, therefore further investigate the importance of identifiers in decompiled binaries

Ahmed and Devanbu use the CodeXGLUE dataset to train their model. The multilingual aspect is achieved by mixing samples from different programming languages in training set to train the model in all languages simultaneously. The single multilingual model is then tested on the test set of the different programming languages in the CodeXGLUE dataset. Using this approach the authors were able to report the highest-scoring model on the CodeXGLUE [30]² Code Summarisation benchmark. Their score was later surpassed by CodeT5 [38]. PolyglotCodeBERT reported an average BLEU4 score of 20.06 over all the programming languages, with the greatest improvements reported in Ruby and JavaScript, the languages with the fewest samples in CodeXGLUE. Recently, DistillCodeT5 reported an even higher score than PolyglotCodeBERT and CodeT5. Unfortunately, as of the writing of this work, DistillCodeT5 is not yet published, and we were unable to find any details about the implementation.

3.3.1 Open Challenges

Although there have been many studies on binary identifier recovery and code summarisation, several aspects have not been properly addressed and investigated. The application of code summarisation methods to decompiled code, has not been addressed by any work at all 3.1. Neutron [27] proposes a Neural Machine Translation solution but does not consider compiler optimisations. Debin [19] and VarBERT recover variable/function names and types. PolyglotCodeBERT proposes a state-of-the-art code summarisation method but only focuses on source code 3.1. Furthermore, some works on binary code fail to take compiler optimisations or stripping into account [27]. We, therefore, investigate the application of code summarisation methods to decompiled stripped and unstripped code. Furthermore, we enable compiler optimisations.

²CodeXGLUE benchmark: <https://microsoft.github.io/CodeXGLUE/>

3. RELATED WORK

Work	Task	Approach	Source	Level	Limitations
Debin	Variable name and type recovery	Extremely Randomized Trees and MAP graphs	Stripped binary	Binary level	Poor generalisability
VarBERT	Variable name and type recovery	Pre-trained and fine-tuned BERT model	Stripped binary	Function level	
FUNCRE	Inlined function recovery	Pre-trained and fine-tuned RoBERTa model	Stripped binary	Function level	
Neutron	Neural Machine Translation	Attention-LSTM	Unstripped binary	Instruction level	No compiler optimisations
PolyglotCodeBERT	Code summarisation	Fine-tuned CodeBERT model	Source Code	Function level	
Our work	Binary Code summarisation	Intermediate-trained and fine-tuned CodeT5 model	Stripped and unstripped decompiled binary	Function level	

Table 3.1: Overview of related work and our proposed solution.

Chapter 4

Approach

The proposed solution starts with a data collection step, where we collect open-source projects. We then compile and decompile these open-source projects. The decompiled functions are aligned with the documentation extracted from the source code. We then process this data to extract descriptive comments and split the data into multiple sets. Finally, we use this data to fine-tune and evaluate a pre-trained CodeT5 model. We also design a few intermediate-training objectives, which are applied to the model before fine-tuning to improve the model performance.

4.1 Data Collection

We require a dataset of decompiled functions labelled with a description to create and assess our solution. This dataset should be relatively large to suit the "data-hungry" nature of our deep-learning models. Furthermore, the dataset needs to feature a diverse set of data representative of our solution's actual real-life use case. To create a large and diverse dataset to train and assess our solution we made use of BinSwarm [3], an existing dataset of aligned decompiled and stripped decompiled functions.¹

Buildswarm starts by collecting C-based projects from Github. The projects are filtered to only include projects that are: Actively being developed, using Travis CI and built for Ubuntu Linux. The projects are built using Docker. The resulting binaries are then copied and stripped, and both the stripped and unstripped binaries are decompiled using Ghidra. The functions are extracted from the stripped and unstripped decompiled code and aligned with the source code. We extract documentation from the original source code to add descriptive comments to this dataset. We depend on the documentation included in the source code by the original authors in the form of single and multiline comments. We align the decompiled functions with

¹BinSwarm; <https://hub.docker.com/r/binswarm/cbuilds>

4. APPROACH

the comments in the source code by using srcML² to extract any documentation located directly before a function signature and then finding the function signature and project name in the decompiled dataset.

A function’s documentation often also contains other details besides the descriptive summary. We found that C projects do not follow a single documentation standard. For example, Javadoc for Java has a short one-line description or summary for each method at the beginning of the multiline comment block. In C, there is no singular documentation standard, so there might not be a single line summary, and we will need to locate it in the comment block automatically. Furthermore, we found that a large number of functions did not have any documentation associated with them at all. To deal with these issues, we did not include any functions that are missing documentation. We devise a few simple rules to extract the summary to find the single sentence description.

A high-level overview of this process is shown in figure 4.1.

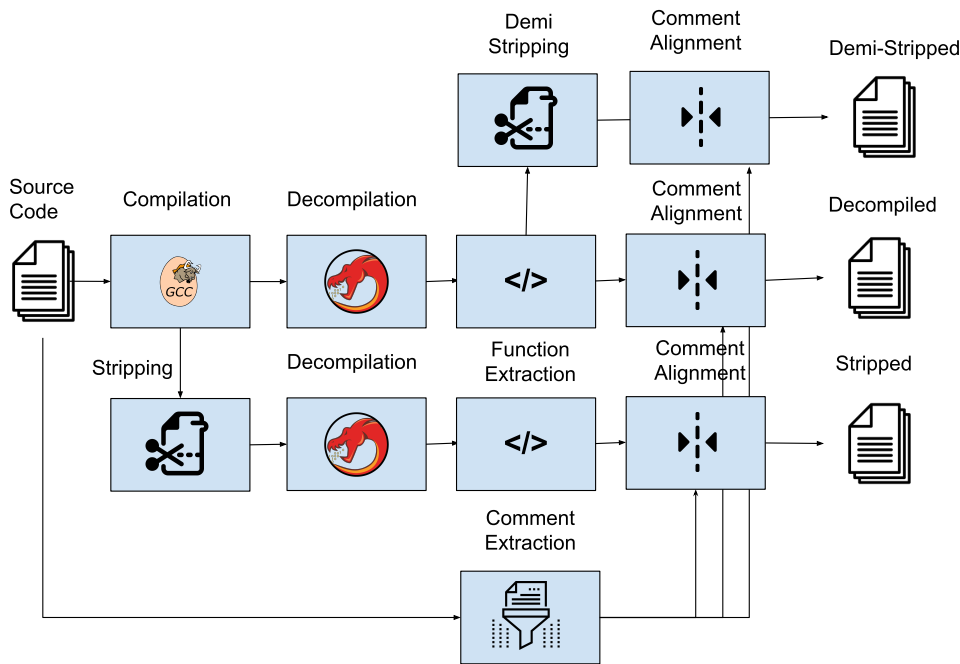


Figure 4.1: Data Collection

From the dataset of decompiled functions, we also create another dataset. We

²srcML: <https://www.srcml.org/>

emulate the process of stripping by removing all the identifiers from the decompiled code and replacing them with placeholders. For clarity, we call this demi-stripped data. Like the stripped dataset, the identifiers are all removed, but the decompiler still had access to the identifiers and could use the symbol table during decompilation. Most importantly, this demi-stripped dataset still has the same structure and control flow as the unstripped decompiled dataset.

4.1.1 Dataset Split

The dataset is split into a train, test and validation set. These sets constitute approximately, 80%, 10% and 10% respectively[25] of the complete dataset. To prevent leakage of vocabulary and code patterns between the sets, we sample the sets in a cross-project manner. This means that an entire project gets assigned to one of the sets, and functions from the same project cannot be assigned to different sets. The different sets should also have a similar distribution of optimisation level and average source-code length.

4.1.2 Duplication

Large corpora of code, like the corpus gathered by BinSwarm, tend to have a high degree of duplication [25]. As a result, snippets of code that are relatively unchanged appear in multiple parts of the corpus. This can be in the form of copied, generic or auto-generated functions. These functions will appear in multiple repositories and might be duplicated across the training and testing data.

Besides exact duplicates, near-duplicates can also occur. Near-duplicates are like exact duplicates, but they differ in a few minor aspects like additional code comments or different function names. While removing exact duplicates is relatively fast and straightforward, removing near-duplicates is much more challenging and computationally intensive [4].

The issue with code duplication in classical code summarization is that the models and tools are supposed to be used to generate summaries for new and unseen code. The evaluation metrics should therefore measure the generalisation of the tool on new samples [4]. Duplicates and near-duplicates are not defined as new samples. A user of such a tool could simply look these samples up. Furthermore, large, high-capacity models like CodeT5 with 220 million [38] or Codex with 12 billion [12] trainable weights, have a large capacity to memorize duplicated code [4]. However, the use case outlined in this work is more akin to deobfuscation. As explained by Allamanis, deobfuscation could be a use case where duplicates are valid and part of the true distribution of the problem[4]. Unfortunately, compiled code contains a lot of duplicate code, and understanding this code is still difficult and essential for

understanding the binary. We, therefore, focus on the model’s performance on code with duplicates, but we also report the deduplicated results.

4.2 Intermediate-Training

The standard CodeT5 model was not pre-trained on any decompiled code. It might therefore be useful to apply additional training steps to ‘teach’ the model the embedding of decompiled and stripped decompiled code. Since we apply these training steps between the pre-training and fine-tuning of the model, we refer to this training strategy as intermediate-training.

To apply and assess other intermediate-training objectives, we train a CodeT5-base model on a predefined objective. Then after that intermediate-training step, we fine-tune the resulting model on our fine-tuning datasets. We essentially apply another training step to the already pre-trained base model. We can then measure the impact of the intermediate-training step on the model’s performance after fine-tuning.

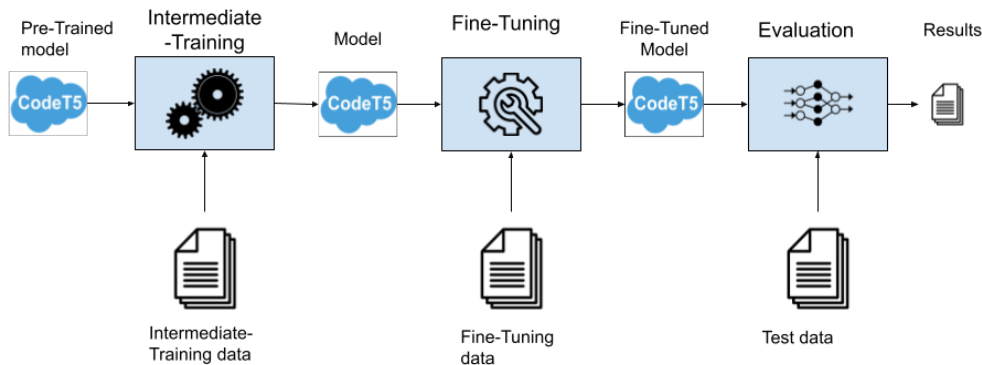


Figure 4.2: Pre-Training

We define several different pre-training objectives. Each of these objectives aims to teach the model the embedding of the identifiers, such that these identifiers can be inferred from the stripped code. For these objectives, we use the relatively large dataset of demi-stripped code. Some samples might be included in both the fine-tuning data and the intermediate-training data, but the model objective will differ. To prevent leakage, we remove the test set of the fine-tuning data from the intermediate-training data.

4.3 Fine-Tuning

The concept of transfer learning, which is utilised in CodeT5, depends on the use of a fine-tuning step to train the pre-trained model on the downstream task. In this case, we make use of the CodeT5-base model, which was trained on mixed upstream tasks by the authors [38].

We fine-tune a pre-trained CodeT5-base model on the constructed dataset. The model is trained on the summarization task as defined in the model. The model is trained on the train set, then evaluated after every epoch on the validation set and finally tested on the test set. During training, the performance of the model is measured using the BLEU-4 metric. BLEU-4 is reported to be unreliable when considering small changes in reported scores [31]. We further evaluate the performance using the EM (exact match), METEOR and ROUGE-L metrics.

Chapter 5

Experimental Setup

5.1 Research Questions

We must create and assess our model to reach our goal of creating an automatic system for decompiled code summarisation. We define research questions which we will answer throughout the thesis. Firstly it is essential to assess the different datasets and set a baseline for each. We can then investigate the impact of different aspects of the data. Finally, we can apply this knowledge to design intermediate-training objectives, further improving performance. This leads to the following research questions:

- **RQ1: How do different input types (source, unstripped decompiled, demi-stripped, stripped) affect the model’s performance? (data-richness effect)**

Firstly, we want to know the impact of the data-richness on the model performance. The different datasets have different degrees of data richness. The source code has all of its identifiers and comments in code. Unstripped decompiled code has no comments and loses many of its identifiers. The decompiler also introduces some noise. Demi-stripped data loses all of the remaining identifiers. Stripped data also has no identifiers and introduces even more decompilation noise.

- **RQ2: What is the impact of data duplication on the model’s performance? (data-duplication effect)**

Secondly, we will evaluate how the model reacts to data duplication, whether the model performance is simply a result of the memorisation of certain examples, or if the performance results from a generalisable understanding of the data.

- **RQ3: To what extent does each aspect of stripped decompiled binaries impact the model’s performance? (data-input study)**

The different datasets each contain different aspects of the original source code. The dataset size, number of decompilation errors and the type and number of recovered identifiers. Which of these aspects are most important for the model performance?

- **RQ4: How do different intermediate-training objectives affect the model’s performance? (model-objective effect)**

Finally, we will apply the insights provided by the previous questions to design new intermediate-training objectives, through which we aim to address the shortcomings of the base model.

5.2 Dataset

To answer the research questions, we construct a diverse and representative dataset. The Buildswarm dataset contains around 1.8m aligned decompiled-sourcecode pairs and 400k aligned stripped-sourcecode pairs. The significant difference is caused by the inherent difficulty finding functions in stripped decompiled code.

From this dataset, we collect any documentation located above the functions using srcML. For example, figure 5.1 shows an example of the source code of a function from the ECDSA library. Above the function definition (line 401), we locate a function block.

```

400
401 <comment type="block"/* Verify an ECDSA signature. */</comment>
402 <function><type><name>int</name></type> <name>secp256k1_ecdsa_verify</name><parameter_list><parameter><dec
403 <decl_stmt><decl><type><name>secp256k1_ge</name></type> <name>q</name></decl>;</decl_stmt>
404 <decl_stmt><decl><type><name>secp256k1_scalar</name></type> <name>r</name></decl>; <decl><type ref="pre
405 <decl_stmt><decl><type><name>secp256k1_scalar</name></type> <name>m</name></decl>;</decl_stmt>
406 <expr_stmt><expr><call><name>VERIFY_CHECK</name><argument_list><argument><expr><name>ctx</name> <oper
407 <expr_stmt><expr><call><name>ARG_CHECK</name><argument_list><argument><expr><name>msghash32</name> <op
408 <expr_stmt><expr><call><name>ARG_CHECK</name><argument_list><argument><expr><name>sig</name> <operat
409 <expr_stmt><expr><call><name>ARG_CHECK</name><argument_list><argument><expr><name>pubkey</name> <oper
410 <expr_stmt><expr><call><name>secp256k1_scalar_set_b32</name><argument_list><argument><expr><operator>
411 <expr_stmt><expr><call><name>secp256k1_ecdsa_signature_load</name><argument_list><argument><expr><name>
412 <return>return <expr><operator></operator><operator>!</operator><call><name>secp256k1_scalar_is_high</
413 <call><name>secp256k1_pubkey_load</name><argument_list><argument><expr><name>ctx</name></expr>
414 <call><name>secp256k1_ecdsa_sig_verify</name><argument_list><argument><expr><operator>&lt;/o
415 </block_content>}</block></function>
416

```

Figure 5.1: Example function with its documentation (truncated)

This documentation can be split into the following classes:

1. Double slash comments, example from Jep:release_utf_char.

```
// release memory allocated by jstring2char
```

These comments are thrown out, as these are generally not used for documentation.

2. Single line comments, example from Nesbox:Curl_mime_read.

```
/* Set mime part remote file name. */
```

We take the entire comment as a description.

3. Multiline comments, example from oftc-ircservices:cs_on_client_join.

```
/**
 * CS Callback when a Client joins a Channel
 * @param args
 * @return pass_callback(self, struct Client *,
→ char *)
 * When a Client joins a Channel:
 * - attach DBChannel * to struct Channel*
 */
```

In this case, we take the first line or sentence.

The data is pre-processed using the Pandas¹ data analysis library as well as the Pandarallel² parallelizability extension for Pandas.

The samples are pre-processed by first extracting the function name and adding it as a column to the data. For the stripped samples, we utilise the alignment with C by extracting the function name from the source code for each sample. Next, the samples are aligned with the comment data using the file name where they reside appended with the function name, for instance: */Repos_Bionic/secp256k1/secp256k1.c:secp256k1_pubkey_load*. We further pre-process the data by removing any newlines from the function body and by unescaping any characters that have been escaped for file safety purposes, such as `>` for `>`.

The samples are split into a train, validation and test set. Each set is collected into a single `.jsonl`³ file.

To answer the second research question, the dataset must be deduplicated. The dataset is deduplicated using a fork⁴ of the near-duplicate-code-detector [4]. We use this tool to compare all the datasets' functions and find clusters of near-duplicate

¹Pandas: <https://pandas.pydata.org/>

²Pandarallel: <https://pypi.org/project/pandarallel/>

³JSON Lines: <https://jsonlines.org/>

⁴Near Duplicate Code Detector: <https://github.com/SERG-Delft/near-duplicate-code-remover>

```
[  
{"repo":"author/project","docstring_tokens":["Tokenized",  
↪ "description"], "code_tokens":["Tokenized", "code"],  
↪ "partition": "train"}  
{"repo":"author/project","docstring_tokens":["Tokenized",  
↪ "description"], "code_tokens":["Tokenized", "code"],  
↪ "partition": "valid"}  
]
```

Figure 5.2: Example entries of a .jsonl file with two dummy entries

functions. We randomly select one function per cluster and discard the rest from the dataset. We use the standard tool configuration as recommended by Allamanis. Of the removed duplicates, we observe that a relatively large number originates from common libraries, such as SQLite⁵, that are packaged with binary programs. Thus a certain amount of duplication is also likely to occur “in the wild”.

5.3 Model Configuration

To first establish a performance baseline, we train a CodeT5-base model on the summarisation task on source C. The baseline is used to compare the decompiled C, stripped decompiled C and the demi-stripped datasets to the source code. For deduplicated code, the results can be compared with the performance of CodeT5-base on the code summarisation task [38], where the performance varies between 15.24 and 26.03 BLEU4 scores depending on the language. Based on this, we set the baseline for a usable summarisation model around a BLEU4 score of 15, with anything lower than a BLEU4 score of 10 essentially unusable. Besides the standard BLEU and EM metrics used by most code summarisation works [17, 38, 30, 2, 1, 25], we also include the METEOR and ROUGE-L metrics. But we found that the metrics mostly aligned, we therefore mainly focus our discussions on the BLEU-4 score.

A grid search of the optimal settings was infeasible from a time perspective, so we performed training mainly using the recommended settings from the CodeT5-base model [38]. We double the source length for the decompiled, stripped, and demi-stripped code to 512 tokens instead of the standard 256 tokens used for the source code. We compensate for the fact that the average length of decompiled

⁵SQLite: <https://www.sqlite.org/index.html>

⁶It is not recommended to use Ghidra versions before 10.1 since these versions have not been patched against a Log4J RCE

Package	Version
Nvidia drivers	510.60.02
cuda	11.6
numpy	1.22.2
tensorboard	2.8.0
torch	1.9.0+cu111
transformers	4.16.2
tree-sitter	0.20.0
Ghidra	10.0.4 ⁶

Table 5.1: The most important packages and their versions

code is almost twice as long as the source code. We utilised two different servers for training 5.3. We trained the model on either an NVIDIA RTX3080 with 10GB of VRAM or an NVIDIA GTX 1080ti with 11GB of VRAM. The authors of CodeT5 used an NVIDIA A100 GPU with 40G of VRAM for fine-tuning [38]. To compensate for the lack of memory, we reduced the batch size to 2, which was the maximum length that still fit both GPUs 5.2.

	CodeT5-base	Our Settings
Source length	256 Tokens	256/512 Tokens
Target length	128 Tokens	128 Tokens
Max epochs	15	15
Patience	2	2
Batch Size	32	2
Vocabulary Size	32100	32100

Table 5.2: Model configuration of the base model and our used settings

	Server 1	Server 2
CPU	Intel XEON E5-2620	AMD Ryzen Threadripper 3990X
Cores	16 (32 threads)	64 (128 threads)
RAM	192GB	128GB
GPU	Nvidia GTX 1080TI	Nvidia RTX 3080
VRAM	11GB	10GB
Storage	7.3TB HDD	1TB NVME SSD

Table 5.3: Hardware used for training and evaluation

5.4 Manual Evaluation

To investigate the third research question, namely the data-input study, we compare the results of the different datasets to see the influence of the different aspects of code on the model performance. Furthermore, we observe that there could be two principal reasons for a sample to be malformed. Firstly, Ghidra can fail to decompile the function correctly. Secondly, during the data collection phase, the comment might not have been appropriately parsed, which results in an incorrect description of the function. To investigate this influence on the stripped model’s performance, we randomly sample 25 high and low-performing samples (in terms of BLEU-4 score) and manually analyse the decompiled code and the description.

5.5 Intermediate-Training

We constructed three intermediate-training objectives, namely Translation, Deobfuscation and Span Prediction, to address the model-objective research question.

5.5.1 Translation

The first defined intermediate-training task is a Neural Machine Translation task. In this code-to-code task, the model has to translate the source code from one programming language to another [30]. We implement a translation from demi-stripped to unstripped decompiled code in our case. Note that, by construction, the only difference between decompiled and demi-stripped code is the lack of identifiers in the demi-stripped code. Figure 5.3 shows an example of a training sample. The input field denotes the input to the model, and the output field depicts the expected model output. We task the model with ”translating” a decompiled function back to the source code. Note that besides the missing identifiers, the structure is also slightly different.

5.5.2 Deobfuscation

The second defined task is a deobfuscation objective. In this code-to-text objective, we task the model with predicting the identifiers in demi-stripped code. Recall that in the demi-stripped code, all identifiers are masked with meaningless placeholders, where duplicate identifiers are assigned the same placeholder. The model will have to output a map of the placeholders to their original value. While the model’s output is somewhat textual and not code, it is not precisely natural language. Figure 5.4 shows an example of a training sample.

```

repo: NLnetLabs/ldns
input: "undefined8 MASK0 ( long param_1 ) {
    undefined8 uVar1;
    if (param_1 != 0) {
        uVar1 = MASK1();
        uVar1 = MASK2(uVar1);
        return uVar1;
    }
    return 0;
}"
target: "uint8_t ldns_rr_label_count (const ldns_rr * rr ){
    if (!rr) {
        return 0;
    }
    return ldns_dname_label_count (ldns_rr_owner(rr));
}"

```

Figure 5.3: Translation intermediate training objective

```

repo: NLnetLabs/ldns
input: "undefined8 MASK0 ( long param_1 ) {
    undefined8 uVar1;
    if (param_1 != 0) {
        uVar1 = MASK1();
        uVar1 = MASK2(uVar1);
        return uVar1;
    }
    return 0;
}"
target: "{MASK0: ldns_rr_label_count, MASK1:
ldns_rr_owner, MASK2: ldns_dname_label_count}"

```

Figure 5.4: Deofbuscation intermediate training objective

5.5.3 Span Prediction

Finally, we define a Span Prediction objective. In this code-to-text objective, we task the model with recovering the identifiers from demi-stripped code. Although, unlike the DOBF objective, every identifier (even matching identifiers) is assigned unique placeholders, the model has to output the assignment of the placeholders in

5. EXPERIMENTAL SETUP

a form closer to natural language and puts more emphasis on duplicated identifiers which might be more critical. Figure 5.5 depicts an illustration of a training sample.

```
repo: NLnetLabs/ldns
input: "undefined8 MASK0 ( long param_1 ) {
    undefined8 uVar1;
    if (param_1 != 0) {
        uVar1 = MASK1();
        uVar1 = MASK2(uVar1);
        return uVar1;
    }
    return 0;
}"
target: "MASK0 ldns_rr_label_count MASK1
ldns_rr_owner MASK2 ldns_dname_label_count"
```

Figure 5.5: Span-detection intermediate training objective

Chapter 6

Results

In this chapter, the results of the experiments are presented, and the results are grouped by research question.

6.1 RQ1: Data-richness study

We collect around 40k stripped-description pairs from the dataset and around 480k decompiled-description and C-description pairs.

The performance of the CodeT5-base model on each of the datasets is presented in table 6.2. The metrics are calculated for each individual sample from the test set, and the average score is presented in the table.

	BLEU-4	EM	METEOR	ROUGE-L
C	36.97	25.56	38.34	40.92
DecomC	33.26	20.20	34.65	37.79
Demi	21.69	13.10	21.22	23.33
Stripped	9.53	3.41	8.53	10.43

Table 6.1: Result of fine-tuning CodeT5-base on the different datasets

We found that the C and DecomC models generally produced good summaries, evident by the BLEU-4 scores over 30. The summaries produced by the demi-stripped model were substantially worse, but most were still very usable, evident by the BLEU-4 score above 20. The stripped model mainly produced unusable summaries, as evidenced by the BLEU-4 score below 10. However, most sequences produced by the model were grammatically and syntactically correct and had some meaning. These could have easily passed for a summary but not for the targeted function. For example, the model produced the output “Unload a C library” against

the reference “Destroy getline object and nullify its pointer”, this output is completely meaningless in the context of the targeted function.

Furthermore, we observed some examples where the model produced a relatively good summary which would likely be very useful but differed heavily from the ground truth. This caused the output to be scored poorly, and the model would suffer a penalty during training. For example, the model produced the output “qsort an array of values by type” against the reference “Sort variables by type”, which results in a low BLEU-4 score, despite being more descriptive.

6.2 RQ2: Data-duplication study

After deduplication, we are left with around 218k decompiled-description pairs and only 7.5k stripped-description pairs. The performance of the base model on each of the datasets is presented in table 6.2:

Deduplicated	BLEU-4	EM	METEOR	ROUGE-L	Δ BLEU-4
C	28.17	14.82	30.60	33.51	8.80
DecomC	19.09	4.68	21.12	24.55	14.17
Demi	11.52	2.24	11.31	13.81	10.17
Stripped	7.03	0.91	6.15	7.49	2.50

Table 6.2: Result of fine-tuning CodeT5-base on the deduplicated datasets and the difference with the baseline

We find that the influence of deduplication on the our model’s performance is relatively small on source code, of only 14%. Duplicate have a relatively large impact on the decompiled (43%) and demi-stripped (47%) code. Of the removed duplicates, we observe that a relatively large number originates from common libraries that are packaged with binary programs.

6.3 RQ3: Data-input study

We find that the performance on decompiled code is slightly lower than source code 6.1, while the performance of demi-stripped code is much higher than stripped code. To control for the impact of the dataset size, we reduce the size of the dataset of the demi-stripped code to match the stripped dataset and fine-tune a CodeT5-base model using the same setup.

We find that the dataset size does not sufficiently explain the significant difference between the demi-stripped and stripped performance. To further investigate this, high and low performing stripped samples were investigated. We randomly

	Samples	BLEU-4	EM	METEOR	ROUGE-L
Demi	40k	17.48	8.22	17.22	19.68
Stripped	40k	9.53	3.41	6.6	7.61

Table 6.3: Comparison between a reduced Demi-stripped and Stripped CodeT5-base model

select 25 samples above and 25 below a certain BLEU-4 score threshold. We set this threshold at a BLEU-4 score of 50.

	Decompilation Failure	Bad Description
High	4/25	6/25
Low	8/25	7/25

Table 6.4: Number of samples which were badly decompiled and which had a badly mined description respectively

We find that the number of inaccurate descriptions is similar (6 vs 7) between the high and low-scoring samples. However, on the flip side, the number of decompilation failures is much higher (4 vs 8). Furthermore, we found that all the decompiled code generally had very few recovered symbols, making it very syntactically poor compared to actual programming languages.

6.4 RQ4: Model-objective study

The results of the intermediate-training objectives are presented in table 6.5. Since the source code and unstripped code perform well, we only focus on improving the performance of the stripped and demi-stripped code. The intermediate-training objectives are baselined against the results from RQ1.

	Stripped				Demi			
	BLEU4	EM	METEOR	ROUGE.L	BLEU4	EM	METEOR	ROUGE.L
Baseline	9.53	3.41	8.53	10.43	21.69	13.10	21.22	23.33
TRANS	5.22	0.92	4.81	5.44	20.74	11.72	20.30	22.37
DOBF	9.98	3.49	8.81	11.17	23.23	12.35	22.77	24.93
SPAN	9.42	3.15	8.64	10.68	22.18	11.47	21.19	24.45

Table 6.5: Result of fine-tuning CodeT5-base after intermediate-training

6. RESULTS

Adding the translation intermediate-training objective did not yield higher scores in either stripped or demi-stripped code across all metrics. We found that the deobfuscation objective resulted in substantially higher scores in both stripped and demi-stripped code across the metrics. The only exception being the EM rate. The span prediction intermediate-training objective yielded mixed results, only improving the METEOR and ROUGE-L scores for stripped code and BLEU-4 and ROUGE-L for the demi-stripped code.

Chapter 7

Discussion

In this chapter, we will provide an analysis and reflection on the results of the experiments. Furthermore, we will discuss the threats to validity and the implications of this work.

7.1 Interpreting the Results

We found a relatively large difference between the number of recovered decompiled and stripped decompiled functions. This can likely be attributed to the fact that Ghidra struggles a lot more with recovering stripped functions. Recall that the symbol table commonly contains information regarding the location and name of functions. When this table is dropped, the start- and endpoints of functions are hard to infer by automatic tools, especially since many functions get inlined, and *JUMP* instructions replace *CALL* instructions. Besides from difficulties in demarcating functions, it is also difficult to align the associated source code function with the decompiled function. With unstripped code, the function name remains, meaning the functions can be aligned using the name. We attempted to utilise an existing solution by Alves-Foss and Song called Jima [6] to find function boundaries. Jima is the current state-of-the-art tool for function boundary detection in stripped binaries. The tool is implemented as a plugin for Ghidra, but in our experiments, we find no statistical difference between the base performance of Ghidra and Jima on our own dataset. The difficulties in extracting stripped functions, make training and applying a model to stripped binaries challenging.

The model produced many instances where the output was grammatically correct and resembled an accurate summary but is meaningless in the context of the targeted function. This shows that the model, or more specifically the decoder, knows the output language well. This is likely because of the pre-training, which included several natural language objectives [38], and the fine-tuning, which is exclusively focused on natural language.

Duplicates have a relatively significant impact on performance. Removing duplicates from the dataset puts the model’s performance more in line with other deduplicated datasets, such as the CodeXGlue dataset used for CodeT5. Removing duplicates has a more considerable impact on decompiled code than on source code. As noted previously in Chapter 4, duplicates are part of the problem space. We, therefore, consider them in the other experiments.

We find a relatively small difference in performance between source code and decompiled code (without demi-stripping). This indicates that in-function-comments and variable names are relatively unimportant for the model performance. Although Ahmed and Devanbu observed that identifiers might be more important than syntax in the code-summarisation task, we can further conclude that the function name is explicitly essential for model performance. Removing just the function name from the decompiled samples, as opposed to removing all identifiers in demi-stripping, results in slightly higher performance than demi-stripped code, which indicates a very high dependence on the name of the function in the code-summarisation task.

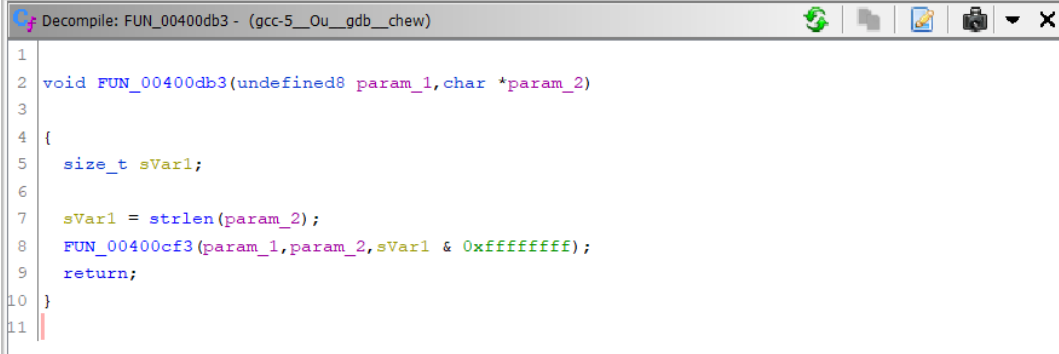
The prediction of function names from the function body is an already defined task, called ”extreme summarisation”, discussed in Chapter 2. While extreme summarisation will help recover function names for stripped functions, which can aid understanding, it has limited applicability to unstripped code, which still retains its function names. In Appendix B we report on a few experiments on this task. We find that, similar to the regular summarisation task, the model struggles with stripped code due to decompiler issues, so we decided not to pursue this avenue any further.

Stripped code performs significantly worse than the demi-stripped code, even when the dataset size is matched. This indicates that the decompilation failures, which occur more with stripped code, also greatly impact model performance.

Our choice of stripper might also influence the performance of the model. We find that our stripped code does not benefit from Ghidra’s decompiler as much as other examples. For example, the decompiled stripped binaries used by David et al. contain much more details than our samples. The sample shown in figure 7.1 is a binary compiled using GCC and optimisation level zero.

We observe that Ghidra’s decompiler manages to recover a variable name and an external call to the `strlen` function. However, in our stripped samples, we do not observe this behaviour, none of the library functions is recovered, and Ghidra predicts no variable names even in samples compiled using `-O0`.

From the intermediate-training objectives, we find that the translation task lowers performance. There could be two explanations for this: Firstly, the model is intermediately-trained on a PL-to-PL objective, while the fine-tuning objective is a PL-to-NL objective. This causes a so-called mismatch in training objectives. Recall that as discussed in Chapter 2 Transformers encode the input into embedding space using an encoder. The decoder will then transform this embedding into the output.



```

Decompile: FUN_00400db3 - (gcc-5__Ou__gdb__chew)
1
2 void FUN_00400db3(undefined8 param_1, char *param_2)
3
4 {
5     size_t sVar1;
6
7     sVar1 = strlen(param_2);
8     FUN_00400cf3(param_1, param_2, sVar1 & 0xffffffff);
9     return;
10 }
11

```

Figure 7.1: Short decompiled function from the GNU Debugger [14] decompiled by Ghidra

In this case, this mismatch in objectives causes the decoder to be weakened in PL-to-NL tasks by the intermediate-training objective. During the pre-training phase, which is applied by Wang et al., the objectives are a mix of NL-to-PL, PL-to-PL and PL-to-NL tasks. This allows a single model to be flexible and to be applied to different tasks.

The second explanation for the lower performance after TRANS intermediate-training is that the target length is also different. While the output for code summarisation is at most 12 tokens long, Neural Code Translation has a target length of 512 tokens. In general, longer target sequences make it difficult for the Transformer model to apply the attention gradient over the entire sequence length properly.

The DOBF and SPAN intermediate-training objectives do yield better performance. Both of these objectives solve the objective mismatch issue and are PL-to-NL objectives. The DOBF performed slightly higher, which could be explained by the shorter target length since it had no repeated identifiers.

7.2 Threats to validity

This section will cover the threats to validity. These threats are split into three categories. The internal threats cover threats to the study’s validity, i.e. whether the conclusions are valid within the confines of our experimental setup. The external threats cover threats to our study’s applicability (generalisability) to situations outside the confines of our experiments. Lastly, the construct validity section covers how well our work covers and measures the intended constructs.

7.2.1 Internal

Noise The training and evaluation data contains a significant amount of noise, either in the form of badly decompiled functions or incorrect documentation. While machine learning models (and specifically NLP models) should be able to handle noisy data, this might introduce some bias into the models.

Inductive bias The base CodeT5 model on which we applied our intermediate-training and fine-tuning strategies will include some bias which would then be transferred to our model.

Lucky sets The randomly selected test set might not represent the data's real distribution.

Data collection Only functions that decompile (Ghidra produces any output) and are commented are represented in the data. This is most apparent in the stripped dataset, where we can only recover a small fraction of the total number of functions.

7.2.2 External

Resisting analysis This work only focuses on stripping as a means of resisting binary analysis, other techniques like control flow or data obfuscation are also used to prevent reverse engineering.

Stripping techniques In this work a very bipolar notion of stripping was used; the strip utility utilised removes any and all identifiers from the binary after compilation. Other techniques, which remove some identifiers before compilation, will result in different decompiled codes with some identifiers left behind by the compiler.

Target Architecture We only considered binaries and projects that were compiled to the x86/x64 architecture. Other architectures like ARM, and other programming languages could provide other results.

Open/Closed-Source The data gathered for the experiments in this thesis, were exclusively from open-source projects. Decompiling closed-source projects is specifically forbidden by some EULAs and the lack of source code documentation makes it difficult to set a reference summary. However, reverse engineering open-source software is not very useful in practise, since the source-code is readily available. Closed-source software might have a different data distribution and will present other challenges like obfuscation.

7.2.3 Construct

Mono-operation bias This work only explores a single state-of-the-art model and only focuses on NLP techniques. Other models, or other techniques, might be more successful at this task.

Mono-method bias This work exclusively focuses on generating code summaries from functions to help REs understand binaries. While the code summarisation task is well defined in the source-code domain, it might not be well suited for binary analysis.

Suitability of metrics As discussed in Chapter 2, the metrics used in this thesis do not capture semantic meaning. BLEU-4 and ROUGE-L have no semantic capabilities and METEOR only has a limited number of word-based rules.

7.3 Broader Impacts

In this thesis, we propose a novel solution to aid reverse engineers in their work. Among many use cases, this solution could help malware analysts to understand novel malware and its weaknesses quickly. The software can be analysed to find possible vulnerabilities and malicious payloads. The source code can be reconstructed for old binaries for which the source code is lost. Finally, it could help in the process of patching binary programs, known as micropatching.

The proposed solution applies a well-known and well-studied task from the software engineering research field to the binary reverse engineering domain. We have proposed a model for one of the many different applications of NLP to code. Other existing and extensively studied applications, like defect detection and code repair, could also be applied to decompiled code.

While the proposed solution has some limitations, especially stripped code, we hope our work inspires other researchers to improve our models further or propose their own solutions. For this, and in the spirit of open science, we have also published our research data. Other researchers can improve our models by designing their own pre-training objectives and applying them to an already trained model. Furthermore, researchers could also use the provided datasets to train and evaluate their own models.

7.4 Ethical Considerations

7.4.1 Automation Bias

The model that we propose can provide REs with assistance in understanding decompiled binaries. However, the automation bias of such a model should be carefully considered, especially since REs can over-rely on the model outputs. The output might be incorrect and could lead the RE down an incorrect path, leading to the RE requiring a longer time to understand the binary or misunderstanding the binary, thus missing specific critical security faults or incorrectly labelling the binary as non-malicious. The model's output should always be taken as a reference by practitioners and checked for correctness.

7.4.2 Offensive Language

As we have observed in the training data, some of the comments that were mined from Github contain insulting or discriminatory statements, such as figure 7.2. Note that this comment has been removed in the most recent version. The prevalence of offensive language in Software Engineering communities such as Github is a known issue [13]. The training data might contain such samples so that this same language might be reflected in the model output.

```
//WTF IS THIS!!!  
//Are you really trying to prevent the analyzed binary  
//from doing anything that would cause it to segfault irl?  
//WHY?  
//      - <Author removed>
```

Figure 7.2: Example of an offensive comment from radareorg/radare2-bindings/radare2/libr/anal/esil.c,

7.4.3 Computational Costs

Training these models requires a non-negligible amount of computational resources. Therefore, we attempted to carefully design our experiments to limit the number of GPU hours. We also selected a pre-trained model, which allows us to skip the expensive pre-training step. Compared to other state-of-the-art LLMs, CodeT5 is a relatively small model with only 220M parameters compared to CodeX, which has 12B parameters [12]. Furthermore, we release our trained models so that the community can avoid repeated training.

7.4.4 Malicious Use

As with much research in the field of binary reverse engineering, this work could be used for malicious purposes. Malicious practitioners could use this work to RE binaries and extract protected intellectual property. Furthermore, this work could be used to analyse binaries for faults and vulnerabilities, such that they can be developed into exploits for use by malicious actors. Any reverse engineer aided by the methods proposed in this work should carefully consider their effort's legal and ethical aspects.¹ Furthermore, any vulnerabilities discovered with the help of these methods should be disclosed responsibly to the owner without needlessly putting the security of users at risk.

¹Electronic Frontier Foundation: <https://www.eff.org/issues/coders/reverse-engineering-faq>

Chapter 8

Conclusion and Future Work

To conclude the thesis, we will first summarise the answers to the research questions posed in Chapter 5 based on our findings and discussion. We will then discuss possible directions for future work based on this thesis.

8.1 Summary

We find that source and unstripped decompiled code performed relatively well, while demi-stripped performed significantly worse. Stripped performs even worse, and the resulting model mainly was unusable. We observe that duplicates have a relatively significant impact on the model performance. Removing these duplicates puts the model performance in line with other source-code datasets. This shows that our models are not only reproducing previously encountered summaries but are also able to create new summaries and have a deeper understanding of the underlying patterns in the data. Finally, we discover that the loss of identifiers causes a drop in performance and that the introduction of decompilation faults in stripped code has a large impact on the model performance.

We find that the model's performance can be improved using intermediate-training objectives. Mainly, we observe that a Deobfuscation intermediate-training objective improves model performance across the board. However, we also find that the effectiveness of the intermediate-training objective is dependent on the fine-tuning objective, as an objective mismatch could cause a drop in performance.

8.2 Future Work

This work solely focuses on the C language. These same techniques could be applied to other compiled languages. Of particular interest is the Go language. Much of the novel malware is written in Go.

8. CONCLUSION AND FUTURE WORK

Besides the exploration of other languages, other stripping techniques could also be explored. We solely focused on the stripping tool included in Linux, but other stripping techniques, some of which are less strict than the Linux implementation, could also be explored.

Furthermore, we focus solely on the Ghidra decompiler, which is free to use. Other paid decompilers could also be used for the decompilation step. IDA-Pro¹ features more advanced function identification methods as F.L.I.R.T.² and Lumina³, which could help with function extraction in stripped binaries.

We found that Ghidra struggles to decompile stripped functions and generally adds minimal syntax or identifiers to the function, which is why exploring these methods on disassembled code might see more success. Disassembled code should have fewer decompilation errors and mistakes. However, using models like CodeBERT and CodeT5 would probably not see much success, as these models are trained on source code. Decompiled code is similar to source code by design, but disassembled code has a relatively limited but very different vocabulary, which could cause Out-Of-Vocabulary issues. One would therefore need to completely retrain models from scratch on data in the disassembled data.

Another solution would be to keep using these methods but add an additional module to determine whether a stripped sample is a well-decompiled function or if a decompilation failure has occurred. Then, the improperly decompiled functions could be skipped, and the RE would only receive the high-quality output.

The output of Ghidra could also be enhanced using a model like DIRE [23] or Debin [19] to recover more identifiers before passing the function to the summarisation model. This method would, however incur an additional performance penalty since it needs to decompile the binaries and recover the identifiers using one of these models. Furthermore, any mistakes and biases introduced by those models would be transferred to our model.

The use of BLEU-4 as a scoring method also introduces some issues. The currently employed scoring methodology does not take semantics into account, meaning that the model could produce perfectly valid and usable descriptions. However, since they do not match the baseline syntax, the score will be very low, and the model will be penalized. Instead of using BLEU-4, METEOR and ROUGE-L scores to score the candidate against the baseline, a semantic method like SentenceBERT [18] could also be employed.

Finally, this work could be developed into a Ghidra plugin. Using Ghidra's scripting engine, extracting the functions from the decompiled code and infer the comments using one of our trained models would be relatively straightforward. To

¹IDA-Pro: <https://hex-rays.com/ida-pro/>

²F.L.I.R.T: https://hex-rays.com/products/ida/tech/flirt/in_depth/

³Lumina: <https://hex-rays.com/products/ida/lumina/>

prevent the constant loading and unloading of the model in the GPU and the requirement for a GPU, the models could even be made permanently available online through an API. The script can then insert the comments above the decompiled functions in the decompiler window. The script would then be run when Ghidra processes a binary to aid the RE in understanding the binary.

Bibliography

- [1] Toufique Ahmed and Premkumar Devanbu. Learning code summarization from a small and local dataset, 2022. URL <https://arxiv.org/abs/2206.00804>.
- [2] Toufique Ahmed and Premkumar Devanbu. Multilingual training for software engineering. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1443–1455. IEEE, 2022.
- [3] Toufique Ahmed, Premkumar Devanbu, and Anand Ashok Sawant. Learning to find usage of library functions in optimized binaries, 2021.
- [4] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, Athens Greece, October 2019. ACM. ISBN 978-1-4503-6995-4. doi: 10.1145/3359591.3359735. URL <https://dl.acm.org/doi/10.1145/3359591.3359735>.
- [5] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code, 2016. URL <https://arxiv.org/abs/1602.03001>.
- [6] Jim Alves-Foss and Jia Song. Function boundary detection in stripped binaries. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19*, page 84–96, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359825. URL <https://doi.org/10.1145/3359789.3359825>.
- [7] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. Variable name recovery in decompiled binary code using constrained masked language modeling. *arXiv preprint arXiv:2103.12801*, 2021.

- [8] Sandrine Blazy and Stéphanie Riaud. Measuring the robustness of source program obfuscation: Studying the impact of compiler optimizations on the obfuscation of c programs. page 123–126, 2014. doi: 10.1145/2557547.2557577. URL <https://doi.org/10.1145/2557547.2557577>.
- [9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [10] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Comput. Surv.*, 48(4), May 2016. ISSN 0360-0300. doi: 10.1145/2896499. URL <https://doi.org/10.1145/2896499>.
- [11] Casey Casalnuovo, Earl T Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. A theory of dual channel constraints. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 25–28, 2020.
- [12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL <https://arxiv.org/abs/2107.03374>.
- [13] Jithin Cheriyan, Bastin Tony Roy Savarimuthu, and Stephen Cranefield. Towards offensive language detection and reduction in four software engineering communities, 2021. URL <https://arxiv.org/abs/2106.02245>.
- [14] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), nov 2020. doi: 10.1145/3428293. URL <https://doi.org/10.1145/3428293>.

-
- [15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. URL <https://arxiv.org/abs/1810.04805>.
- [16] Christiane Fellbaum. Wordnet. In *Theory and applications of ontology: computer applications*, pages 231–243. Springer, 2010.
- [17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.
- [18] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. Semantic similarity metrics for evaluating source code summarization. *arXiv e-prints*, pages arXiv–2204, 2022.
- [19] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.
- [20] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, apr 2016. ISSN 0001-0782. doi: 10.1145/2902362. URL <https://doi.org/10.1145/2902362>.
- [21] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, 2008.
- [22] M Tim Jones. Optimization in gcc. *Linux journal*, 2005(131):11, 2005.
- [23] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.
- [24] Alon Lavie and Michael J Denkowski. The meteor metric for automatic evaluation of machine translation. *Machine translation*, 23(2):105–115, 2009.
- [25] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics:*

- Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3931–3937, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1394. URL <https://aclanthology.org/N19-1394>.
- [26] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.
- [27] Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: an attention-based neural decompiler. *Cybersecurity*, 4:5, 03 2021. doi: 10.1186/s42400-021-00070-0.
- [28] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL <https://aclanthology.org/W04-1013>.
- [29] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.
- [30] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. 2021. doi: 10.48550/ARXIV.2102.04664. URL <https://arxiv.org/abs/2102.04664>.
- [31] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. *Reassessing Automatic Evaluation Metrics for Code Summarization Tasks*, page 1105–1116. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450385626. URL <https://doi.org/10.1145/3468264.3468588>.
- [32] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
- [33] Jürgen Schmidhuber, Sepp Hochreiter, et al. Long short-term memory. *Neural Comput*, 9(8):1735–1780, 1997.
- [34] Shi, Ensheng Wang, Yanlin Du, Lun Chen, Junjie Han, Shi Zhang, Hongyu Zhang, Dongmei Sun, and Hongbin Sun. On the evaluation of neural code summarization. ICSE, 2022.

- [35] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.
- [36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6000–6010, 2017.
- [37] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. An observational investigation of reverse engineers’ process and mental models. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI EA ’19, page 1–6, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359719. doi: 10.1145/3290607.3313040. URL <https://doi.org/10.1145/3290607.3313040>.
- [38] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.
- [39] Zhuo Zhang, Wei You, Guan hong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676. IEEE, 2021.

Appendix A

Glossary

In this appendix, we give an overview of frequently used terms and abbreviations.

BLEU: BiLingual Evaluation Understudy

CNN: Convolutional Neural Network

COTS: Commercial Off-The-Shelf

DOBF: Deobfuscation

EM: Exact Match

IR: Intermediate Representation

LLM: Large Language Model

LSTM: Long Short-Term Memory

MASK: Mask Prediction

ML: Machine Learning

MLM: Masked Language Modelling

NL: Natural Language

NLP: Natural Language Processing

NMT: Neural Machine Translation

PL: Programming Language

RE: Reverse Engineer or Reverse Engineering

A. GLOSSARY

RNN: Recurrent Neural Network

seq2seq: Sequence-to-sequence

SOTA: State-Of-The-Art

SPAN: Span detection

TL: Transfer Learning

Appendix B

Extreme Summarisation

In this appendix we report on the application of the extreme summarisation task on our dataset.

B.1 Methodology and Experimental Setup

We apply the parameters and setup as provided by Ahmed and Devanbu for the extreme summarisation task.¹ Note that the authors use their setup for the summarisation task and only change the maximum target length of the output to 10 tokens. We fine-tune a CodeBERT [17] model for 5 epochs, we chose this limit for brevity as we found that all models converged during the third or fourth epoch as can be observed in figure B.1. After every epoch we evaluate the model using the validation set, and finally the model is tested using the test set.

To prepare the dataset, we use the same pipeline as shown in figure 4.1. Instead of extracting the comment data, the name of the functions is extracted from their aligned source code. These function names will function as reference. Since missing or unalignable documentation is no longer an issue, we are able to collect significantly more decompiled function-function name pairs. Our dataset consists of 2.1m decompiled and 630k stripped samples. Recall that unstripped decompiled functions still retain the function name after decompilation, so we selectively strip away the function name from the function definition and any recursive calls in the function bodies.

The dataset is split into a train, test and validation set. Similarly to the regular summarisation task we split them in a cross-project manner, with the sets constituting 80%, 10% and 10% of the complete dataset respectively. We followed the recommendations by LeClair and McMillan on the dataset construction.

¹Model and parameters: <https://zenodo.org/record/5670434>

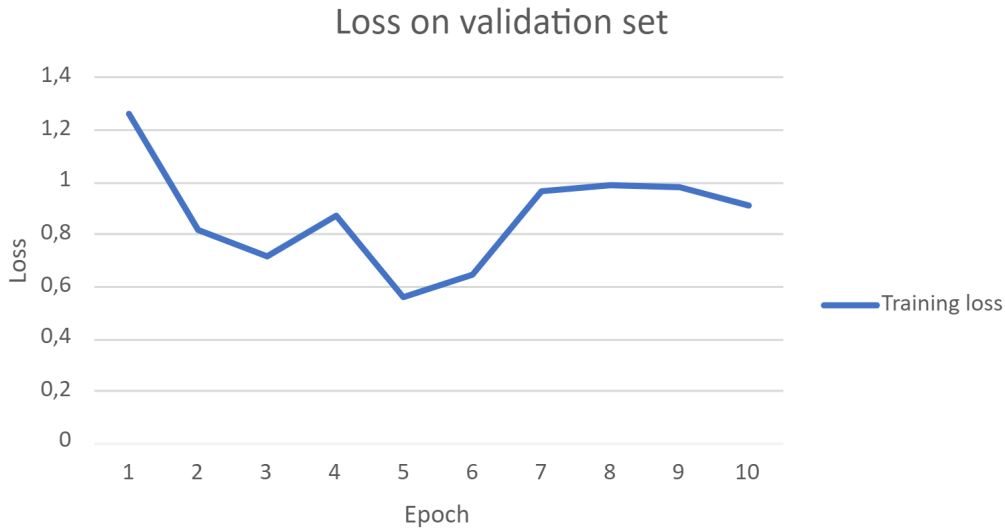


Figure B.1: Loss during CodeBERT training on stripped data

Since the output of the model is so short and generally contains up to 3 tokens [5], BLEU would be unfit as a metric. Similarly to Allamanis et al. and Ahmed and Devanbu, we chose to use the F1-score.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} * 100$$

Every function name is broken up into subtokens, the name "popStack" would therefore be tokenised into "pop" and "stack". The precision and recall are then calculated by comparing the tokens in the reference and the model prediction. Note that this does not take the order of the tokens into account.

Ahmed and Devanbu report F1 scores ranging from 24 to 54 and F1 scores ranging from 0.41 and 0.54 with regular CodeBERT and PolyglotCodeBERT respectively. Note that these models were trained and evaluated on the CodeXGLUE [30] dataset which is deduplicated, while our own dataset is not.

B.2 Results

We find that the results resembled the results from the regular code summarisation task. The CodeBERT model achieved an F1 score of 20.6 and 5.45 on the stripped and unstripped code respectively. Manual assessment of the produced samples shows that the model for unstripped code could produce some usable samples, but the stripped model was unable to produce almost anything of value, as can be observed in table B.1.

Reference	Decom	Stripped
main	main	main
sha 256 transform	md 5 process block	sha 1 process block
get ins len	op nd num reg s	op nd get size
r id storage foreach	queue fore ach	hash find
map init	pro g init	c se g open

Table B.1: Sample of CodeBERT extreme summarisation model output