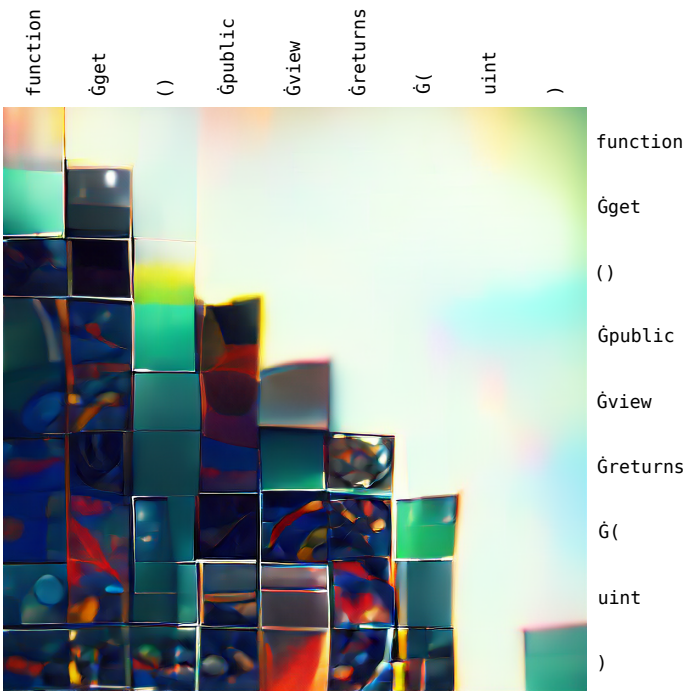
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

André Storhaug

Secure Smart Contract Code Synthesis with Transformer Models

Master's thesis in Computer Science
Supervisor: Jingyue Li
July 2022



Synthetic image of transformer attention weights. Source: André Storhaug

NTNU
Norwegian University of
Science and Technology

André Storhaug

Secure Smart Contract Code Synthesis with Transformer Models

Master's thesis in Computer Science
Supervisor: Jingyue Li
July 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Writing Smart Contracts (SCs) are hard. Writing secure SCs is even harder. Automatic code generation is by many considered the ”holy grail” in the field of computer science. Recent advances in transformer models have shown great potential in the area of synthesizing code from code comments. However, it is just as important *how* these models are best put to use. In this thesis, it is investigated: how to automatically generate Smart Contract code with transformer-based language models, by inputting comments to guide the code generation? Due to the monetary and immutable nature of blockchain, security in SCs is of uttermost importance. This thesis also investigates: how to generate secure Smart Contract code with transformer-based language models? A design science research approach is adopted for answering these research questions. For automatically synthesizing SC code, the 6 billion parameter model GPT-J by EleutherAI is fine-tuned on SC code. For this task, the currently largest dataset of real SC code is constructed, containing 186,397 contracts. To evaluate the comment-aid approach to generate code, the original code was used as the ground truth. This code was then compared with the generated code, and their differences were measured using the BiLingual Evaluation Understudy (BLEU) score. Results of the evaluation show that this approach results in a BLEU score of 0.557, which is beyond the state-of-the-art. For generating secure Smart Contract code with transformer-based language models, a novel method named security conditioning is proposed. From both automatic and manual evaluation of the technique, the evaluation results show that security conditioning produced secure code.

Sammendrag

Å skrive smart kontrakter (SK) er vanskelig. Å lage dem sikre er enda vanskeligere. Automatisk kodegenerering regnes av mange som den "hellige gral" innen datavitenskap. Nylige fremskritt innen transformer modeller har vist stort potensiale innen syntese av kode fra kodekommentarer. Imidlertid er det mist like viktig å vite *hvordan* disse modellene best kan brukes. I denne oppgaven undersøkes det: hvordan generere SK kode automatisk med transformerbaserte språkmodeller, ved bruke kommentarer for å veilede kodegenereringen? På grunn av den økonomiske og uforanderlige naturen til blokkjede teknologi, er sikkerhet i SKer av ytterste viktighet. Denne oppgaven undersøker også: hvordan generere sikker SK-kode med transformatorbaserte språkmodeller? En designvitenskapelig forskningstilnærming brukes for å besvare disse forskningsspørsmålene. For automatisk syntetisering av SK-kode er 6 milliarder parametermodellen GPT-J av EleutherAI finjustert på SK-koden. For denne oppgaven er det for øyeblikket største datasettet med ekte SC-kode konstruert, som inneholder hele 186,397 kontrakter. For å evaluere tilnærmingen til kommentarhjelp for å generere kode, ble den opprinnelige koden brukt som fasit. Denne ble deretter sammenlignet med den genererte koden, og forskjellene deres ble målt ved å bruke BiLingual Evaluation Understudy (BLEU)-skåren. Resultatene av evalueringen viser at denne tilnærmingen resulterer i en BLEU-skår på 0,557, som er bedre enn nåværende sate-of-the-art. For å generere sikker SK-kode med transformatorbaserte språkmodeller, foreslås en ny metode kalt security conditioning. Fra både automatisk og manuell evaluering av teknikken viser evalueringsresultatene at security conditioning produserte sikker kode.

Acknowledgement

I wish to express my deepest gratitude to my supervisor, Professor Jingyue Li, for all the help and guidance throughout the entire project. I also want to acknowledge Ms. Tianyuan Hu for her help with vulnerability analysis. Finally, I want to acknowledge all the love and support from my family - my parents, Synnøve and Ove; and my sisters, Maria, Viktoria and Helene. This work would not have been possible without them.

This work is supported by the Research Council of Norway (No.309494).

André Storhaug, Trondheim 19.05.2022

Contents

Figures

Tables

Code Listings

Acronyms

API Application Programming Interface. 28

AST Abstract Syntax Tree. 14, 15, 56

BERT Bidirectional Encoder Representations from Transformers. 3, 15

bfloat16 Brain Floating Point. 22, 26, 42

BLEU BiLingual Evaluation Understudy. vii, viii, xiv, xv, xvii, 9, 15, 16, 56–61, 67–69, 72, *Glossary:* BiLingual Evaluation Understudy

BPE Byte-Pair Encoding. 41

CLM Casual Language Modeling. xvi, 24, 42, 43, 50

DAO Decentralized Autonomous Organization. 19

DSR Design Science Research. 19

ELMo Embeddings from Language Models. 15

EVM Ethereum Virtual Machine. xvii, 57, *Glossary:* Ethereum Virtual Machine

float16 Half-precision Floating-Point. 26

GPT General Pre-trained Transformer. 3, 15

IR Intermediate Representation. xvii, *Glossary:* Intermediate Representation

LoC Lines of Source Code. 30

LSTM Long Short-Term Memory. 15

ML Machine Learning. 2

NFT Non Fungible Tokens. xvii, 10, *Glossary:* Non Fungible Tokens

NLTK Natural Language Toolkit. 35

NVMe Non-Volatile Memory Express. xviii, 26, *Glossary:* Non-Volatile Memory Express

OOM Out of Memory. xviii, 42, *Glossary:* Out of Memory

PCA Principal Component Analysis. 37

PCFG Probabilistic context-free grammar. 14

RNN Recurrent Neural Network. 3

RoPE Rotary Position Embedding. 22, 41

SC Smart Contract. vii, viii, xiii, xvi, 2, 3, 10–13, 16, 19–21, 24, 25, 28, 29, 33, 41, 42, 48–52, 55–58, 67–73

SMT Satisfiability Modulo Theories. xviii, *Glossary:* Satisfiability Modulo Theories

TFIDF Term Frequency–Inverse Document Frequency. 35

WoS Web of Science. xviii, *Glossary:* Web of Science

ZeRO Zero Redundancy Optimizer. 25, 26, 42

Glossary

BiLingual Evaluation Understudy Metric for automatically evaluating machine-translated text. vii, viii, xiv, xv, 9, 15, 16, 56–61, 67–69, 72

docstring Python function documentation strings. 15

Ethereum Virtual Machine The runtime environment for transaction execution in Ethereum. 57

F1 Harmonic mean of precision and recall. 14

Non Fungible Tokens A type of token that is unique. 10

Non-Volatile Memory Express A standard hardware interface for solid state drives (SSDs) that uses the PCI Express (PCIe) bus. 26

Out of Memory An often undesired state of computer operation where no additional memory can be allocated. 42

Chapter 1

Introduction

The art of computer programming is an ever-evolving field. The field has transformed from punchcards to writing assembly code. With the introduction of the C programming language, the field sky-rocketed. Since then, many new languages have been introduced, and the art of programming has become a complex and ever-changing field. Today, computer systems are all around us and permeate every aspect of our lives. However, constructing such systems is a hard and time-consuming task. Several tools and methods have been developed to increase the productivity of programmers, as well as to make programming more accessible to everyone.

Recent advancements in large-scale transformer-based language models have successfully been used for generating code. Automatic code generation is a new and exciting technology that opens up a new world of possibilities for software developers. One example is GitHub Copilot [1]. Copilot uses these models to generate code for a given programming language. The tool is based on a deep learning model, named Codex [2] by OpenAI, that has been trained on a large corpus of code. This enables developers to significantly speed up productivity. In addition, it makes programming more accessible to everyone by significantly reducing the threshold for using various programming languages and libraries. Another recent contribution is AlphaCode [3], a code generation tool for generating novel code solutions to programming competitions.

The language models are getting larger and better by the day. However, it is just as important *how* these models are best put to use. Describing functionality is easy. Implementing it in code is hard. Almost every coding language supports some form of code comment. These are normally created to explain the implemented code after the code is written. Recent works [2, 4] have leveraged the power of transformers to automatically generate code from comments. This has the potential to greatly lower the threshold for non-developers to leverage programming, while also increasing efficiency due to a simpler developing process. However, none of these works investigates how to best write these comments for guiding code generation. Further, for evaluating these systems, they only consider generating code from comments in isolation. This is rarely the case in a

real-world setting because developers write code and use comments to explain the code. This thesis investigates a comment-aided approach to generating code, meaning the use of the existing code and comments as the input. For generating code, this work does not exclude the opportunity for developers to type in code and comments in combination with the automatically generated code.

A machine learning model is only as good as the data it is fed. These large-scale transformers need huge amounts of data to be trained. Normally, this data is collected from all available open source code. A problem with this is that a lot of this code contains security problems. This can be everything from exposed API keys, to exploitable vulnerabilities. Autocomplete tools like GitHub Copilot must therefore be used with caution [2].

To better secure automatic code generation, this thesis also purposes a novel approach for use of ML models in large-scale transformer-based code generation pipelines to ensure secure generated code. To demonstrate the approach, this thesis will focus on generating secure code for Smart Contracts (SCs) (Solidity). Smart Contracts (SCs) have an exceptionally high demand for security, as vulnerabilities can not be fixed after a contract is deployed. Due to most blockchains' monetary and anonymous nature, they pose as a desirable target for adversaries and manipulators [5]. Further, SCs tends to be rather short and simple, making it a good fit for generated code. The main research questions addressed in this thesis are:

- How to automatically generate Smart Contract code with transformer-based language models, by inputting comments to guide the code generation?
- How to generate secure Smart Contract code with transformer-based language models?

The specific contributions of this thesis are as follows:

- The currently largest Smart Contract (SC) dataset.
- Novel comment-aided code generation using the fine-tuned transformer-based language model to generate Smart Contract code.
- Novel method to secure code generation.
- Identification of open issues, possible solutions to mitigate these issues, and future directions to advance the state of research in the domain.

The rest of this paper is organized as follows. Chapter 2 describes the background of the project. The research related to this document is commented in Chapter 3. Chapter 4 describes the research methodology employed for implementing the research questions. The implementation and the results are presented in Chapter 5. The implementations are then evaluated in Chapter 6. The findings and results from the implementation and evaluation are discussed in Chapter 7. Finally, Chapter 8 concludes the thesis, presenting final remarks and future works.

Chapter 2

Background

This chapter introduces the necessary background information for this study. First, a thorough description of the Transformer model is provided in Section 2.1. Section 2.2 explain the different metrics used in this thesis, followed by a brief introduction to blockchain technology in Section 2.3. Then, the concept of Smart Contracts (SCs) and the most popular SC vulnerabilities are described in Section 2.4.

2.1 Transformer

A transformer is a deep learning model. It is designed to process sequential data and adopts the mechanism of self-attention. The Transformer model architecture was introduced in 2017 by Vaswani *et al.* [6]. Unlike more traditional attention-based models such as Recurrent Neural Networks (RNNs), transformers do not include any recurrence or convolutions. This allows the model to process the entire input all at once, solely relying on attention. It solves the vanishing gradient problem of recurrent models, where long-range dependencies within the input are not accurately captured. It also allows the model to be significantly more parallelized, making training on huge datasets feasible. Because of this, pre-trained systems such as Bidirectional Encoder Representations from Transformers (BERT) [7] and GPT [8] were developed. These models are pre-trained on a large corpus of text, such as Wikipedia Corpus and Common Crawl, and effectively predict the next word in a sentence. Further, the models can be fine-tuned on a new dataset to improve their performance on more specialized tasks.

2.1.1 Architecture

The standard Transformer architecture, as described by Vaswani *et al.* [6] in 2017, is shown in Figure 2.1. The following subsections describe the architecture of the standard Transformer model.

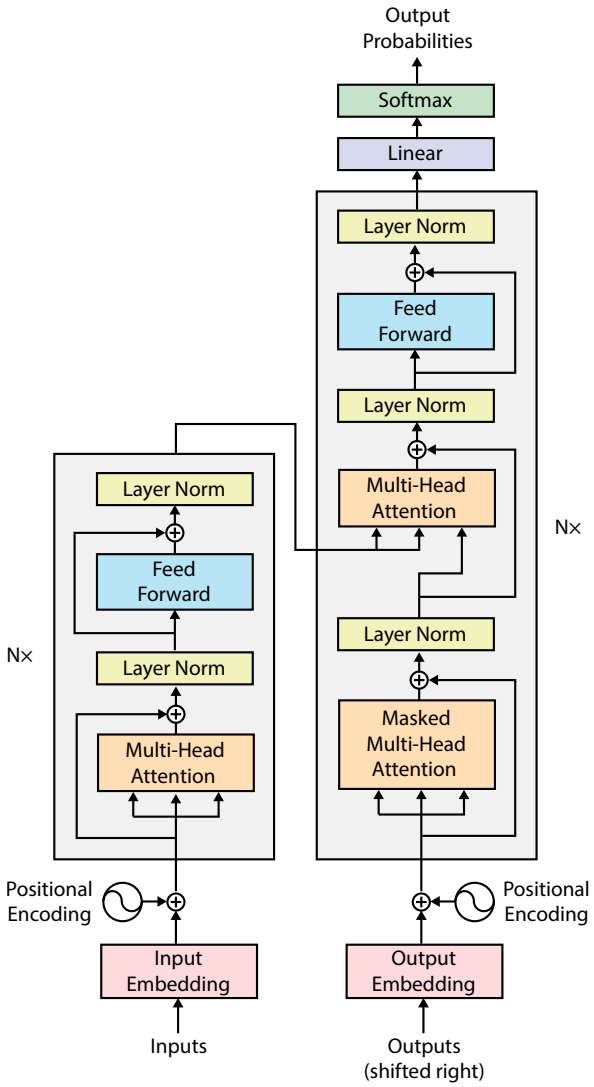


Figure 2.1: Architecture of a standard Transformer Vaswani *et al.* [6]

2.1.1.1 Tokenization

For a Transformer to process the text input, the text is first tokenized. Tokenization is the process of breaking a sequence of text into a sequence of tokens. For example, the sentence *I am a sentence.* is tokenized into the words "*I*", "*am*", "*a*", "*sentence*", and ".". The tokenization process is usually done by a tokenizer. Specifically, the transformer uses a byte pair encoding tokenizer.

2.1.1.2 Embedding and Positional Encoding

After the input text is tokenized, the next step for the model is to understand the meaning and position of the token (word) in the sequence. This is achieved by an Embedding layer and a Positional encoding layer. The results of these two layers are combined.

Two embedding layers are used. The Input Embedding layer is fed the input sequence. The Output Embedding layer accepts the target sequence after shifting the target to the right by one position and inserting a start token at the first position. The embedding layers produce a numerical representation of the input sequence, mapping each token to an embedding vector.

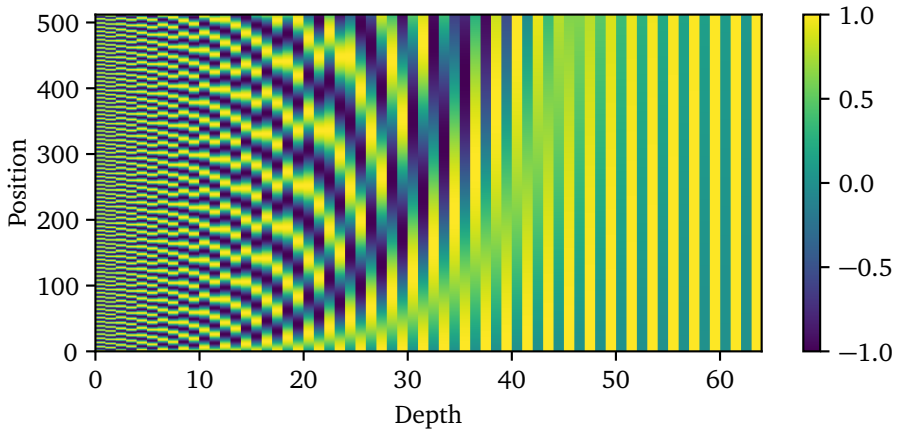


Figure 2.2: The 64-dimensional positional encoding for a sentence with the maximum length of 512. Each row represents an positional encoding vector.

The positional encoding is generated by a sinusoidal positional encoding layer. This layer is fed the sequence length and produces a sinusoidal positional encoding vector. This is illustrated in Figure 2.2, where each row corresponds to one sinusoidal positional encoding vector. The positional encoding vector is then added to the embedding vector.

2.1.1.3 Encoder and decoder stacks

A Transformer is comprised of two main parts: the encoder and the decoder. The encoder is responsible for encoding the input sequence into a sequence of vectors. It tries to capture information about which parts of the inputs are relevant to each other. The decoder is responsible for decoding the output sequence from the encoder. Along with other inputs, the decoder is optimized for generating outputs. In Figure 2.1, the left and right halves represent the Transformer encoder and decoder, respectively.

The encoder and decoder are both composed of a stack of self-attention layers. This layer allows the model to pay more or less attention to certain words in the input sentence as it is handling a specific word. Each decoder layer has an additional attention mechanism that draws information from the outputs of previous decoders, before the decoder layer draws information from the encodings. Both the encoder and decoder layers contain a feed-forward layer for further processing of the outputs, as well as layer normalization and residual connections.

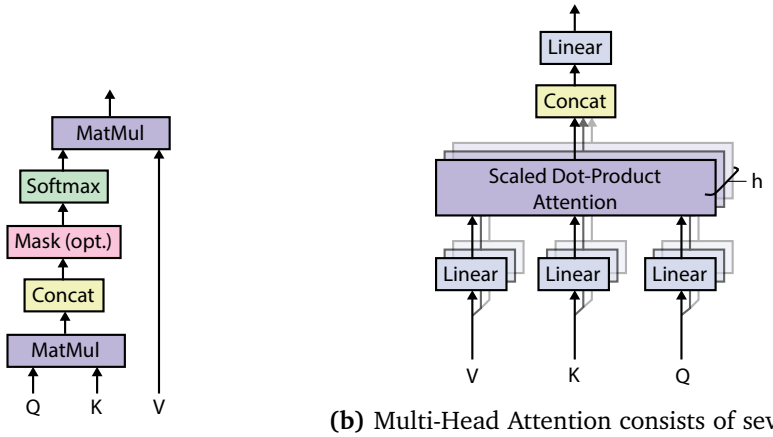
The transformer architecture allows for auto-regressive text generation. This is achieved by re-feeding the decoder the encoder outputs. The decoder then generates the next word in a loop until the end of the sentence is reached. For this to work, the Transformer must not be able to use the current or future output to predict an output. The use of a look-ahead mask solves this. The final output from the transformer is generated by feeding the decoder output through a linear layer and a softmax layer. This produces probabilities for each token in the vocabulary and can be used to predict the next token (word).

The encoder and decoder can also be used independently or in combination. The original transformer model described by Vaswani *et al.* [6] used an encoder-decoder structure. These models are used for generative tasks that also require input, for example, language translation or text summarization. Encoder-only models are used for tasks that are centered around understanding the input, such as sentence classification and named entity recognition. Decoder-only models excel at generative tasks such as text generation.

2.1.1.4 Scaled dot-product attention

The self-attention layer used in each Transformer block is named "Scaled Dot-Product Attention". An overview of the attention layer is shown in Figure 2.3a. The layer learns three weight matrices, query weights W_Q, key weights W_K, and value weights W_V. Each input word embedding is multiplied with each weight matrix, producing a query vector, key vector, and value vector. Self-attention scores are then generated by calculating the dot products of the query vector with the key vector of the respective word (query) that is calculated.

In order to stabilize the gradients during training, the attention weights are divided by the square root of the dimension of the key vectors, $\sqrt{d_k}$. A softmax function is then applied, normalizing the scores to be positive and adding up to 1. Each value vector is then multiplied by the softmax score. The resulting weighted



(a) Scaled Dot-Product Attention.

(b) Multi-Head Attention consists of several attention layers running in parallel.

Figure 2.3: Multi-Head Attention module in Transformer architecture Vaswani *et al.* [6]

value vectors are then summed up and serve as output from the attention layer.

In practice, the attention calculation for all tokens can be expressed as one large matrix calculation, as shown in Figure 2.3a. This significantly speeds up the training process. The queries, keys, and values are packed into the separate matrices Q, K, and V, respectively. The output matrix can be described as:

$$\text{Attention}(Q,K,V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

where the superscript T represent the transpose operation.

2.1.1.5 Multi-head attention

By splitting the query, key, and value parameters in N-ways (logically), each with its separate weight matrix, the performance of the Transformer is increased. This is called multi-head attention, illustrated in Figure 2.3b. It gives the Transformer greater power to encode multiple relationships and nuances for each word. The final attention outputs for the feed-forward network are calculated by concatenating the matrixes for each attention head.

2.1.2 Training

A Transformer model typically undergoes something called self-supervised learning. This is an intermediary between both unsupervised- and supervised learning. This normally conforms to unsupervised pre-training the model on a large set of data. Then, the model is fine-tuned on a (usually) smaller dataset of labeled data.

In contrast to the unsupervised training, where the target sequence comprises the predicted transformer output, the supervised training is done by feeding the

complete input- and target language sequence directly into the Transformer. The input sequence is fed to the encoder, while the target sequence is fed to the decoder.

2.1.3 Inference

For making inference, the Transformer is only fed the input sequence. The encoder is run on the input sequence, and the encoder output is fed to the decoder. Since no encoder output is available at the first timestep, the decoder is fed a special "<start>" token. The decoder output is then fed back into the decoder again. This process is repeated until the decoder output encounters a special "<stop>" token.

2.2 Relevant Metrics

2.2.1 Machine learning performance metric

2.2.1.1 Accuracy

Accuracy is the proportion of correct predictions among the total number of cases processed [9]. Accuracy is formally defined as the proportion of correct predictions among the total number of cases processed, as seen in Equation (2.2).

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.2}$$

where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives.

It is a very common metric used for evaluating the performance of a machine learning model. However, it has to be used with caution, as an overfitted model would report high accuracy.

2.2.1.2 Perplexity

Perplexity is one of the most common metrics for evaluating language models [10]. It is a measure of how variable a prediction model is, and can be defined as the normalized inverse probability of the test set [11]. For a test set with words $W = w_1, w_2, \ldots, w_N$, the perplexity of the model on the test set is:

$$PP(W) = \sqrt[N]{\frac{1}{p(w_1, w_2, ..., w_N)}} \tag{2.3}$$

Perplexity can be interpreted as the weighted branching factor. If we have a perplexity of 10, it means that whenever the model tries to guess the next word, it is as confused as if it had to pick between 10 words. Models with lower perplexity have probability values that are more varied. Meaning, the lower perplexity, the better model [11].

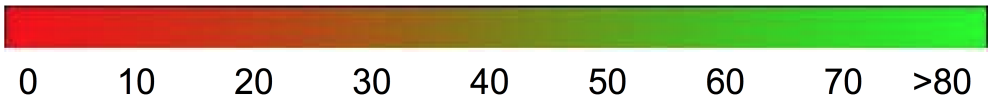


Figure 2.4: Gradient for interpreting BLEU score Lavie [14].

2.2.2 Machine translation performance metrics

2.2.2.1 BLEU

BLEU (BiLingual Evaluation Understudy) by Papineni *et al.* [12] is a metric for automatically evaluating machine-translated text. BLEU scores are between 0 and 1. A value of 0 means there is no overlap with the reference translation, while a value of 1 means that the translation perfectly overlaps. A score of 0.6 or 0.7 is considered the best a human can achieve [12, 13]. The color gradient in Figure 2.4 from [14] can be used as a general scale interpretation of the BLEU score.

The method is based on n-gram matching, where n-grams in the reference translation are matched against n-grams in the translation. The matches are position-independent. The more matches, the higher the score.

For example, consider the following two translations:

Candidate: on the mat the cat sat.
Reference: The cat is on the mat.

The unigram precision $(p_1) = 5/6$

However, machine translations tend to generate an abundance of reasonable words, which could result in an inaccurately high precision. To combat this, BLEU uses something called modified precision[12]. The modification consists of clipping the occurrence of an n-gram to the maximum number the n-gram occurs in the reference. These clipped precision scores (p_n) are then calculated for n-grams up to length N, normally 1-grams through 4-grams. They are then combined by computing the geometric average precision, as shown in Equation (2.4). In addition, positive weights w_n are used, normally set to $w_n = 1/N$.

$$\text{Geometric Average Precision } (N) = \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{2.4}$$

BLEU also introduces a brevity penalty for penalizing translations that are shorter than the reference:

$$\text{Brevity Penalty} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{2.5}$$

The final BLEU score is then computed as:

$$\text{BLEU} = \text{Brevity Penalty} \cdot \text{Geometric Average Precision Scores } (N) \tag{2.6}$$

2.2.3 String metric

2.2.3.1 Jaccard index

The Jaccard index [15] is also known as the Jaccard similarity coefficient. It is a statistic used for gauging the similarity of sample sets. It is defined as the size of the intersection divided by the size of the union of the sets, as shown in Equation (2.7).

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.7}$$

The Jaccard index ranges from 0 to 1. The higher the number, the more similar the two sets are.

The Jaccard index can for example be used as a measure of how similar two text strings are. For this, the strings are simply converted into sets of n-grams. The n-grams are then compared using the Jaccard index.

2.3 Blockchain

Blockchain technology was popularized by Satoshi Nakamoto in 2008 with his publication of the article "Bitcoin: A Peer-to-Peer Electronic Cash System". He introduced the formal idea of a peer-to-peer electronic cash system based on blockchain. This made it possible for users to conduct transactions without any need for a central authority. A blockchain is a growing list of records linked together with the help of a cryptographic hash. Each of these records is called a block. The blocks contain a cryptographic hash of the previous block, transactional data, and a timestamp. Since all blocks contain the hash of the previous block, they end up forming a chain. To tamper with a block in the chain, this also requires altering all subsequent blocks. Because of this, Blockchains are resistant to modification. The longer the chain, the more secure it is.

2.4 Smart Contract

The term "Smart Contract" was introduced with the Ethereum platform in 2014. A Smart Contract (SC) is a program that is executed on a blockchain. This enables non-trusting parties to create an *agreement*. SCs have enabled several interesting new concepts, such as Non Fungible Tokens (NFT) and entirely new business models. Ever since Ethereum's introduction of SCs, the platform has kept its position as one of the most popular SC blockchain platforms. Ethereum is an open, decentralized platform that allows users to create, store, and transfer digital assets. Solidity is the primary programming language that is used to write these SCs

for Ethereum. Solidity is compiled down to bytecode, which is then deployed and stored on the blockchain. Ethereum also introduces the concept of gas. Ethereum describes gas as follows: "It is the fuel that allows it to operate, in the same way that a car needs gasoline to run [16]". The gas is used to pay for the cost of executing a SC. This also protects against malicious actors spamming the network [16]. The gas is paid in Wei, which is the smallest denomination of Ether. Due to the immutable nature of blockchain technology, once a smart contract is deployed, it cannot be changed. This can have serious security implications, as vulnerable contracts can not be updated.

2.4.1 Smart Contract Security Vulnerabilities

There are many vulnerabilities in Smart Contracts (SCs) that can be exploited by malicious actors. Throughout the last years, an increase in the use of the Ethereum network has led to the development of SCs that are vulnerable to attacks. Due to the nature of blockchain technology, the attack surface of SCs is somewhat different from that of traditional computing systems. The Smart Contract Weakness Classification (SWC) Registry [1] collects information about various vulnerabilities. Following is a list of the most common vulnerabilities in Smart Contracts:

2.4.1.1 Integer Overflow and Underflow

When an arithmetic operation reaches the maximum or minimum size of a certain data type, an integer overflow or underflow occurs. For example, adding or multiplying two integers may result in a value that is unexpectedly small. Considering the opposite, subtracting from a small integer may result in an unexpectedly large positive value. For example, an 8-bit integer addition $255 + 2$ might result in 1.

2.4.1.2 Transaction-Ordering Dependence

There is no guarantee on the execution order of transactions in blockchain systems. A miner can influence the outcome of a transaction due to its own reordering criteria. For example, a transaction that is dependent on another transaction to be executed first may not be executed. This can be exploited by malicious actors, and is called transaction-ordering dependence, or TOD.

2.4.1.3 Broken Access Control

Access Control issues are common in most computer systems, not just smart contracts. However, because of the monetary nature and transparency of SCs, properly enforcing access controls are essential. Broken access control can for example occur due to wrong visibility settings of functions. This gives attackers a relatively straightforward way to access contracts' private assets. However, the bypass methods are sometimes more subtle. For example, in Solidity, reckless use

[1] `https://swcregistry.io`

of `delegatecall` in proxy libraries, or use of the deprecated `tx.origin` might result in broken access control. Code listing 2.1 shows a simple Solidity contract where anyone can trigger the contract's self-destruct, which makes the code vulnerable. Due to its severity, unprotected self-destructs are also recognized as a separate vulnerability, named unprotected suicide.

Code listing 2.1: Access control vulnerable Solidity Smart Contract code

```
1  contract SimpleSuicide {
2      function sudicideAnyone() {
3          selfdestruct(msg.sender);
4      }
5  }
```

2.4.1.4 Timestamp Dependency

If a Smart Contract is dependent on the timestamp of a transaction, it is vulnerable to attack. A miner has full control over the execution environment for a SC. If the SC platform allows for SCs to use the time defined by the execution environment, this may result in a vulnerability. An example of vulnerable use is a timestamp used as part of the conditions to perform any critical operation (e.g., sending ether) or as the source of entropy to generate random numbers. Hence, a malicious miner could gain an advantage by choosing a suitable timestamp for a block he is mining. Code listing 2.2 shows an example Solidity SC that contains this vulnerability. Here, the timestamp (the `now` keyword on line 10) is used as a source of entropy to generate a random number.

Code listing 2.2: Timestamp Dependency vulnerable Solidity Smart Contract code

```
1  contract Roulette {
2      uint public prevBlockTime; // One bet per block
3      constructor() external payable {} // Initially fund contract
4
5      // Fallback function used to make a bet
6      function () external payable {
7          require(msg.value == 5 ether); // Require 5 ether to play
8          require(now != prevBlockTime); // Only 1 transaction per block
9          prevBlockTime = now;
10         if(now % 15 == 0) { // winner
11             msg.sender.transfer(this.balance);
12         }
13     }
14 }
```

2.4.1.5 Reentrancy

Reentrancy is a vulnerability that occurs when a SC calls an external contract. Most blockchain platforms that implement SCs provide a way to make external contract calls. In Ethereum, an attacker may carefully construct a SC at an external address that contains malicious code in its fallback function. Then, when a contract sends funds to the address, it will invoke the malicious code. Usually, the malicious code triggers a function in the vulnerable contract, performing operations not expected by the developer. The name "reentrancy" comes from the fact that the external malicious contract calls a function on the vulnerable contract and the code execution then "reenters" it. Code listing 5.1 shows a Solidity SC function where a user can withdraw all the user's funds from a contract. If a malicious actor creates a contract that calls the withdrawal function several times before completing, the actor would successfully withdraw more funds than the current available balance. This vulnerability could easily be eliminated by moving the updating of the balance on line 4 to above the transferring of funds on line 3.

Code listing 2.3: Reentrancy vulnerable Solidity Smart Contract code

```
1 function withdraw() external {
2     uint256 amount = balances[msg.sender];
3     require(msg.sender.call.value(amount)());
4     balances[msg.sender] = 0;
5 }
```

Chapter 3

Related work

This chapter presents related research in the field of source code synthesis. Section 3.1 presents various techniques for code synthesis. The section begins with presenting some of the earlier techniques, followed by surveying more recent and state-of-the-art code synthesis techniques. In Section 3.2 works related to bias in language models are presented.

3.1 Code synthesis

Code synthesis is the task of generating code from a given specification. One of the earlier classical works used a probabilistic Probabilistic context-free grammar (PCFG) [17]. Hindle *et al.* [18] investigated whether code could be modeled by statistical language models. In particular, the authors used an n-gram model. They argue that "programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties". They found that code is more predictable than natural languages. DeepCoder by Balog *et al.* [19] focused on solving programming competition-style problems. They trained a neural network for predicting properties of source code, which could be used for guiding program search.

3.1.1 Code synthesis based on code semantics

Programs can also be synthesized by leveraging the semantics of the code. Alon *et al.* [20] purposes a tool named code2vec. It is a neural network model for representing snippets of code as continuously distributed vectors, or "code embeddings". The authors leverage the semantic structure of code by passing serialized Abstract Syntax Trees (ASTs) into a neural network. Code2seq [21] builds on the works of Alon *et al.* [20] which focuses on natural language sequence generation from code snippets. The authors use an encoder-decoder LSTM model and rely on ASTs for code snippets. The model is trained on three Java corpuses small, medium, and large, achieving a F1 score of 50.64, 53.23, and 59.19, respectively. However, the model is limited to only considering the immediately sur-

rounding context. Pythia by Svyatkovskiy *et al.* [22] is able to generate ranked lists of method and API recommendations to be used by software developers at edit time. The code completion system is based on ASTs and uses Word2vec for producing code embeddings of Python code. These code embeddings are then used to train a Long Short-Term Memory (LSTM) model. The model is evaluated on a dataset of 15.8 million method calls extracted from real-world source code, achieving an accuracy of 92%.

3.1.2 Code synthesis based on transformers

Inspired by the success of large natural language models such as ELMo (Embeddings from Language Models) [23], GPT (General Pre-trained Transformer) [8], BERT (Bidirectional Encoder Representations from Transformers) [7], XLNet [24], and RoBERTa [25], large-scale Transformer models have been applied in the domains of code synthesis. Feng *et al.* [26] proposes a new approach to code synthesis by training the BERT transformer model on Python docstring paired with functions. The resulting 125M parameter transformer model, named CodeBERT [26], achieves strong results on code-search and code-to-text generation. The authors also observe that models that leverage code semantics (ASTs) can produce slightly better results. PyMT5 Clement *et al.* [4] is based on the T5 model. The model can predict whole methods from natural language documentation strings (docstrings) and summarize code into docstrings of any common style. For method generation, PyMT5 achieves a BiLingual Evaluation Understudy (BLEU) score of 0.0859 and a F-score of 24.8 on the CodeSearchNet [27] test set. GPT-C by Svyatkovskiy *et al.* [28] is a model based on GPT-2. The 366M parameter-sized model is trained on a code corpus consisting of 1.2 billion lines of source code in Python, C#, JavaScript and TypeScript programming languages. The Python-only model reportedly achieves a ROUGE-L precision of 0.80 and recall of 0.86.

The model complexity of transformers has recently sky-rocketed, with model sizes growing to several tens of billions of parameters. GPT-J is a 6 billion parameter model trained on The Pile, which is an 825GB dataset. The pre-trained version of GPT-J is also publicly available. Codex by Chen *et al.* [2] is a 12 billion parameter model based on GPT. It was trained on 54 million GitHub repositories, and a production version of Codex powers GitHub Copilot [1]. The model solves 28.8% of the problems in the HumanEval dataset [2], while GPT-3 solves 0% and GPT-J solves 11.4%. Google DeepMind's AlphaCode [3] is 41.4 billion parameters and is the first AI to reach a competitive level in programming competitions. AlphaCode was tested against challenges curated by Codeforces [29], a competitive coding platform. It achieved an average ranking of 54.3% across 10 contests. The authors found that repeated sampling on the same problem significantly increased the probability of a correct solution.

Table 3.1: Existing language models.

Refs.	Year	Model	Metrics	Languages	Input	Output
[26]	2020	CodeBERT	BLEU	Python	Code	Docstring
[4]	2020	PyMT5	BLEU	Python	Docstring	Code
[28]	2020	GPT-C	BLEU	Python, C#, JavaScript, TypeScript		
[2]	2021	Codex	Functional correctness	Python	Docstring	Code
[3]	2022	AlphaCode	Functional correctness	Python, C++, Java	Problem description	Code

As can be seen from Table 3.1, most of the models are concerned with Python code. However, none have attempted to generate Smart Contracts (SCs) code. SC code is quite different from most of the other popular languages such as Python, JavaScript, and Java. Investigating how transformer models perform on SC code would give valuable insight into the future of code synthesis. Further, all of the works listed in Table 3.1 that are concerned with text-to-code generation, only consider using comments in isolation. There is therefore a need for an investigation of a code generation approach that uses both comments and code for generating functions.

3.2 Bias in language models

One of the main problems with language models is that they often contain bias [30]. This can be everything from producing gender-specific jobs to favoring a certain race. There have been several works related to mitigating this in language models. However, with varying success. Silva *et al.* [31] tried to mitigate societal bias in text generation using a loss regularizer to "de-bias" a RoBERTa model. However, their approach was not successful and conclude there is a need for more robust bias testing in transformers. Several works are devoted to using adversarial methods to reduce bias. Madras *et al.* [32] propose LAFTER, an adversarial method to modify the training objective based on a desired fairness measure. Zhang *et al.* [33] also tries to reduce bias using adversarial training. However, [33] states that the adversarial training method is hard to get right and is often touchy. Hofstätter *et al.* [34] investigates bias in visual transformer models, as they find existing approaches such as LAFTR unable to maintain high performance. They propose TADeT, a targeted alignment strategy for debasing transformers that aims to discover and remove bias primarily from query matrix features.

In the area of code synthesis, vulnerabilities can be considered a form of bias in language models. However, there seems to be very little research on the security

of synthesized code using transformers. [2] provide a brief discussion of insecure code generated by Codex. However, this investigation was limited to the exploration of the generation of cryptographic functions. Pearce *et al.* [35] acknowledge this gap in research and conduct a vulnerability analysis of GitHub Copilot (based on Codex). They construct a manual dataset of incomplete python and C code that *may* produce a vulnerability. From their analysis of 1689 synthesized Python and C programs, they conclude with approximately 40% are vulnerable. This shows there is a dire need for reducing the number of vulnerabilities generated with language models.

Chapter 4

Research Methodology

This chapter presents the research methodology used in this thesis. Firstly, the research motivation is presented in Section 4.1, followed by the research questions defined for this thesis in Section 4.2. The research method and design are explained in Section 4.3. Then, Sections 4.4 and 4.5 presents the research design for RQ1 and RQ2, respectively. Finally, Section 4.6 presents the various software libraries and hardware used in this thesis.

4.1 Research Motivation

Writing Smart Contracts are hard. Writing secure Smart Contracts is even harder. Automatic code generation is by many considered the ”holy grail” in the field of computer science [36]. Ever since OpenAI introduced its first transformer model in the GPT series, this class of transformers has been touted as the state-of-the-art for text generation. Recent works have applied transformers for code generation and program synthesis, achieving state-of-the-art results. For example, Codex by Chen *et al.* [2] fine-tunes GPT-3 [37] on code data from GitHub. The results are impressive. However, while the models are improving at a staggering rate, it is also important to consider *how* the models should be used. Works such as [2, 4] only consider code synthesis from comments. This is clearly problematic, as users of these systems, developers, do not normally develop code in isolation.

These systems also face many other problems, especially in regards to different biases, for example, gender and security biases. Chen *et al.* [2] describes that because their model (Codex) is trained on open-source code, including "Public code may contain insecure coding patterns, bugs, or references to outdated APIs or idioms.", the model might "synthesize code that contains these undesirable patterns introduce vulnerabilities" [1]. An empirical study by Pearce *et al.* [38] found that almost approximately 40% of the generated code by GitHub Copilot is vulnerable. Security flaws in software can have dire consequences. According to Smith *et al.* [39] the estimated cost of cybercrime for 2020 is over $1 trillion dollars. Due to the monetary nature of blockchain, security flaws are often very severe, as exploits of vulnerabilities often directly result in the loss of funds. One of the

most infamous blockchain attacks was the crowdfunding project Decentralized Autonomous Organization (DAO) hack. This hack resulted in an economic loss worth about 60 million dollars at the time [40]. Further, the immutable nature prevents the possibility of correcting vulnerable code after being deployed.

This thesis tries to address the problems and limitations described above, by answering the research questions defined in Section 4.2.

4.2 Research Questions

The research questions addressed in this thesis are:

RQ1. How to automatically generate Smart Contract code with transformer-based language models, by inputting comments to guide the code generation?

RQ2. How to generate secure Smart Contract code with transformer-based language models?

4.3 Research Method and Design

To best facilitate the answering of the research questions defined in Section 4.2, an Design Science Research (DSR) was selected as the research approach. A DSR focuses on the development and performance of artifacts. For this to be considered research, the work needs to demonstrate academic qualities such as analysis, explanation, argument, justification, and critical evaluation. Further, the work needs to contribute to knowledge in some way [41]. DSR is typically an iterative process that involves five steps [42]: Awareness of Problem, Suggestion, Development, Evaluation, and Conclusion.

- **Awareness of Problem**: is the recognition and formulation of a problem. This might come from multiple sources, such as areas identified by authors for further research, reading about new developments in the industry, from other disciplines, new technological developments, etc. The output of this phase is a proposal for a new research effort, either formal or informal.
- **Suggestion**: directly follows the development of a proposal based on an awareness of a problem. This is the creative step where a tentative idea of how to solve such a problem in a novel way is suggested.
- **Development**: is the actual implementation of the suggested idea. This is the step where the tentative design idea is implemented and produces an artifact. The techniques used for implementation vary with the type of artifact, which could be anything from algorithms to models.
- **Evaluation**: is the evaluation of the artifact. In this step, the artifact's worth is assessed, as well as potential deviations from expectations.
- **Conclusion**: is the final step where the results from the design process are determined to be "good enough". The results are written up. The knowledge

gained is identified, along with any loose ends that might serve as subjects for future research.

For this thesis, Section 4.1 clearly describes the awareness of the problems this thesis aims to solve. This is the motivation behind the new research effort proposed in this thesis, conveyed as research questions defined in Section 4.2. Section 4.4 and Section 4.5 describe a suggestion for how to answer these research questions. Chapter 5 describes the implementation of the suggested solution for the research questions, while Chapter 6 presents an evaluation of the implementation results. Finally, the findings and results are discussed in Chapter 7, and areas suitable for further research are presented in Chapter 8.

4.4 Design for RQ1

This section describes the design for research question 1. For constructing a comment-aided system for automatically generating smart contract code, multiple design steps are needed. First, the design for a code comment analysis is described in Section 4.4.1. Then the following Section 4.4.2 describes the language model selected for use in this thesis. Finally, the design of the fine-tuning process is described in Section 4.4.3

4.4.1 Code comments analysis

Code comments come in different shapes and styles. Solidity, the primary SC language of Ethereum, has no less than 4 different standard comment types. For example, one can use single- or multi-line NatSpec comments. These are comments that can provide rich documentation for functions, return variables and more [43]. An example of this is shown in Code listing 4.1. To provide some insight into how a user can formulate a comment for guiding the code synthesis, a comment analysis is conducted of how these look like in real SC code. Specifically, a clustering analysis of the comments is conducted. Such an analysis can also be used for providing insight into the performance of the models developed in this project.

Code listing 4.1: Example of a NatSpec comment.

```
1 /// @title A token implementation
2 /// @author André Storhaug
3 /// @notice This is an example implementation
4 /// @dev All function calls are currently implemented without side effects
```

4.4.2 Language Model to use

As discussed in Section 3.1.2, there are several available transformer models. However, only a few of them have open-sourced pre-trained weights. Of these,

GPT-J [44] is the largest model that includes code in its pre-training dataset "The Pile", described in Section 4.4.2.1. The research community has found these models to outperform existing open-source GPT systems in qualitative programming evaluations [45]. These findings are further backed by [2]. Because of this, the state-of-the-art generative pre-trained transformer model GPT-J is the language model used in this thesis.

4.4.2.1 The Pile

The Pile [46] is an 825 GiB open source dataset for language modeling. The Pile features many disparate domains, including books, GitHub repositories, webpages, chat logs, and medical, physics, math, computer science, and philosophy papers, making it one of the most extensive and diverse datasets available. Figure 4.1 shows a treemap of the different parts of the dataset. Especially interesting is that 7.59% (about 95.16 GiB) of the Pile is made up code from GitHub. Code from around 192K GitHub repositories are included, all with more than 100 stars and smaller than a gigabyte [47]. Unfortunately, as of February 2022, less than 10 repositories on GitHub contain Solidity code and have over 100 stars [1]. Hence there is a need for a dataset made up of SCs.

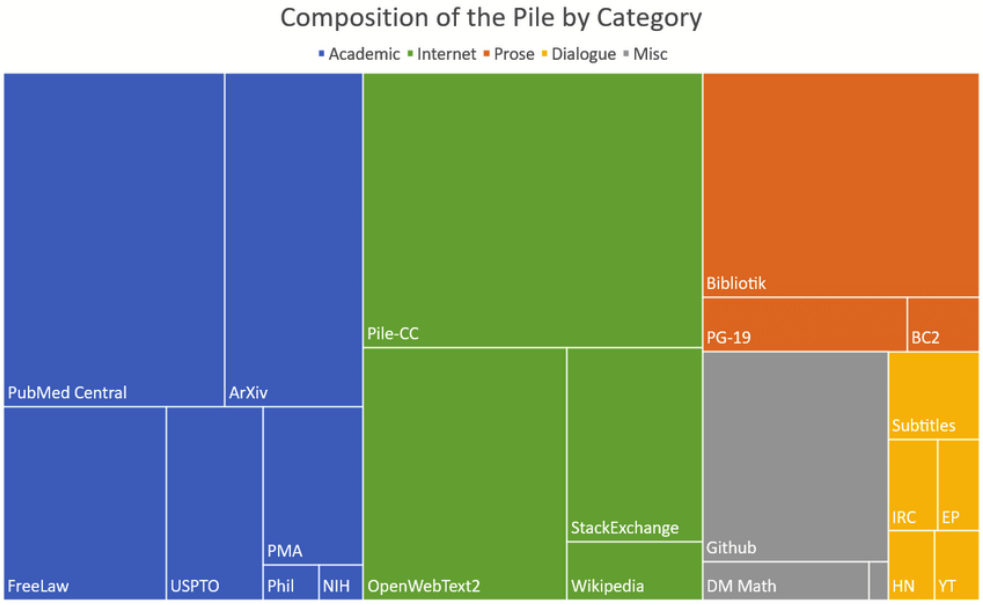


Figure 4.1: Treemap of the Pile components by effective size. Source: [46]

[1] https://github.com/search?o=desc&q=language%3ASolidity&type=Repositories

4.4.2.2 Model architecture

Ever since OpenAI introduced its first transformer model in the GPT series, this class of transformers has been touted as the state-of-the-art for text generation. Their latest model, GPT-3 [37], is their best performing model with 175 billion parameters. However, the model is not openly available at the current time. GPT-J [44] with 6 billion parameters (GPT-J-6B) is currently one of the best open-source alternatives to OpenAI's GPT-3. GPT-J was released in June 2021 by EleutherAI [48], a grassroots collection of researchers working to open-source AI research. The model is trained on the Pile, an 825 GiB diverse, open-source language modeling data set that consists of 22 smaller, high-quality datasets combined together. See section Section 4.4.2.1 for a more detailed description of the Pile.

Being a GPT class transformer, GPT-J uses a decoder-only architecture, as can be seen in Figure 4.2. The GPT-J introduces some notable differences from standard transformer models. Firstly, instead of computing attention and feed-forward layers in sequential order, they are computed in parallel and the results are added together. This decreases communication during distributed training, resulting in increased throughput. Secondly, GPT-J uses Rotary Position Embedding (RoPE) [49] for position encoding. Opposite to sinusoidal encoding used in standard transformer models (see Section 2.1.1.2), this is shown to result in better model quality in tasks with long text [49].

4.4.2.3 Requirements

To load the GPT-J model in float32 precision, one would need at least 2x the model size of CPU RAM: 1x for the initial weights and another 1x to load the checkpoint. So for just loading the GPT-J model, it would require at least 48GB of CPU RAM. To reduce the memory footprint, one can load the model in half-precision.

GPU needs around 40GB of GPU memory to load the model. For training/fine-tuning the model, it would require significantly more GPU RAM. For example, the Adam optimizer makes four copies of the model: model, gradients, average and the squared average of gradients. Hence, it would take 4x model size GPU memory, even with mixed precision as gradient updates are in fp32. Further, this doesn't include the activations and data batches which would require some more GPU RAM. Hence, solutions like DeepSpeed [50] needs to be used for training/fine-tuning such large models.

If a GPU with mixed precision capabilities (architecture Pascal or more recent) is available, one can use mixed precision training with PyTorch 1.6.0 or later, or by installing the Apex library for previous versions. If using an NVIDIA "Ampere" GPU architecture, the Brain Floating Point (bfloat16) floating-point format can be used. Using mixed precision training usually results in 2x-speedup for training with the same final results.

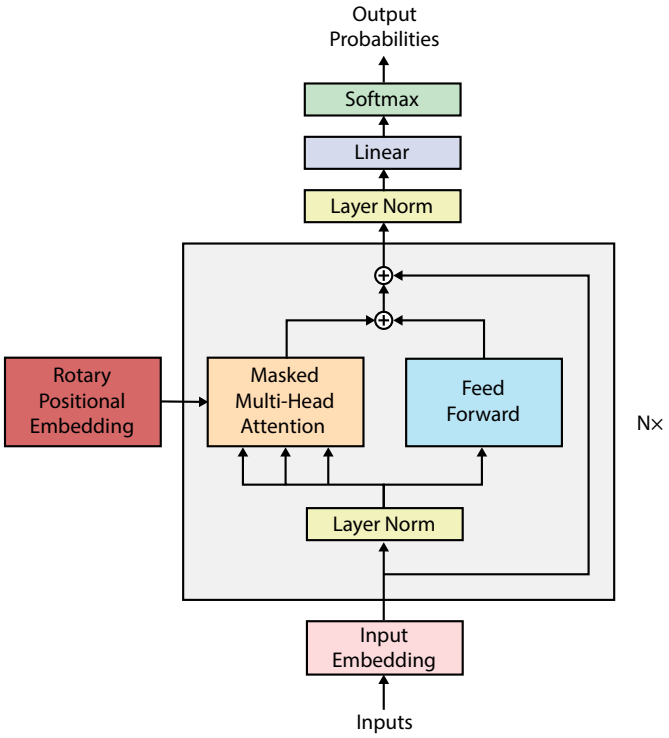


Figure 4.2: Diagram of GPT-J model architecture.

4.4.2.4 Pre-training

Pre-training is defined as "Training in advance". By first training the model on a huge dataset, the model can then be fine-tuned on a much smaller dataset. This is so-called transfer learning. The pre-training procedure used for GPT class models is called Casual Language Modeling (CLM) [8]. The model reads the text input in order and then tries to predict the next word. The model is fed a complete text element (input sequence) all at once, and then internal masking is applied to prevent the model from cheating by looking at future tokens. For more details on the inner workings of the training procedure, see Section 2.1.2.

4.4.3 Fine-tuning design

To refine the pre-trained GPT-J-6B model for generating SC code, the model needs to be fine-tuned on SCs. This should allow the model to adapt its existing knowledge gained from the pre-training procedure to produce high-quality SC code. The fine-tuning procedure used is similar to that of the pre-training procedure. However, instead of using the Pile, a custom dataset of SCs needs to be constructed for use. The dataset then needs to be shuffled, before SCs are fed to the model for training. To ensure the validity of the model's performance, the dataset used needs to be split into separate sets for training, validation and testing. In this thesis, 80% of the data will be used for testing, 10% for validation, and 10% for testing. After the model is fine-tuned, it should be able to auto-regressively generate SCs code. Due to the share size of the selected model (see Section 4.4.2.3), no hyper-parameter optimization was performed. All hyper-parameters were set to their default values, as used for pre-training.

4.5 Design for RQ2

This section describes the design for research question 2. The proposed method for generating secure SC code with transformer models is described in Section 4.5.1. Then, the design of the fine-tuning process is described in Section 4.4.3.

4.5.1 Security Conditioning

When training a large language model on several gigabytes of open-source code, it is safe to assume that large portions of this code are not safe and contains vulnerabilities. For example, Durieux *et al.* [51] analyzed 47.587 real SCs with 9 automatic analysis tools. From these, 97% of the contracts are tagged as vulnerable. This can result in a biased model that may produce a lot of vulnerable code. The idea of this thesis is to use a technique, named security conditioning, to reduce and mitigate this problem.

The security conditioning is done by prepending a special security label to each of the records in the training data. This way, the model can learn to associate these tokens with either secure or vulnerable code. This way, by also using the labels

when generating code, the model may be able to condition whether to produce safe or vulnerable code.

4.5.2 Fine-tuning design

For fine-tuning the pre-trained model with security conditioning, much of the same procedure as in Section 4.4.3 is used. This makes it possible to validate the technique, by comparing the secured model with the "unsecured" model developed in RQ1. The primary difference is that the dataset records need to be labeled as secure or vulnerable. Before training, the dataset is shuffled and the SCs are fed to the model for training. To ensure the validity of the model's performance, the dataset needs to be split into separate sets for training, validation and testing. In this thesis, 80% of the data will be used for testing, 10% for validation, and 10% for testing. To be able to The same hyperparameters used for as in Section 4.4.3 no hyper-parameter optimization was performed. All hyper-parameters were set to their default values, as used for pre-training.

After the model is fine-tuned, it should be able to auto-regressively generate SCs code. To invoke the security conditioning, one only needs to prepend the security label to the input text.

4.6 Technology

Following is an overview of the different technologies applied in this project, both software and hardware.

4.6.1 Software

During the selection of the language modeling library for use in this project, several considerations were made. Firstly, due to the huge size of the model, the library needed to support distributed GPU training. It had to be flexible and scalable, without sacrificing too much on speed. The transformers [52] library by Hugging Face [53] fulfilled these conditions. The library provides flexible and easy-to-use solutions. It also supports integration with DeepSpeed [50], a deep learning optimization library by Microsoft [54] that makes distributed training and inference easy, efficient, and effective. The Hugging Face ecosystem also provides the Datasets and Tokenizers libraries, streamlining and significantly simplifying the use of large datasets.

DeepSpeed. The deep learning optimization library DeepSpeed [50] is used for training. It facilitates both distributed training, mixed precision and gradient accumulation, providing significant speedup of the training process while still being able to fit the model into the GPU memory available. The main workhorse of DeepSpeed is the Zero Redundancy Optimizer (ZeRO) [55]. ZeRO comes with

three incremental optimization stages: stage 1 (ZeRO-1), stage 2(ZeRO-2) and stage 3(ZeRO-3).

- **Stage 1**: partitions the optimizer states across the processes, so each process only updates its partition.
- **Stage 2**: partitions the reduced gradients for updating the model weights, so that each process only retains the gradients corresponding to its own portion of the optimizer states.
- **Stage 3**: partitions the model parameters across the processes. They are automatically collected and partitioned during forward and backward passes.

For training exceptionally large models, DeepSpeed also provides heterogeneous memory technologies based on ZeRO. This includes ZeRO-Offload for ZeRO-2 and ZeRO-Infinity [56] for ZeRO-3. ZeRO-Offload offloads the optimizer memory and computation from the GPU to the host CPU. ZeRO-Infinity is an upgraded version of ZeRO-Offload that also allows for offloading to Non-Volatile Memory Express (NVMe) memory. DeepSpeed ZeRO makes it possible to train trillion parameter models [56]. However, each optimization stage comes with a performance cost, slowing down the training process. DeepSpeed also provides support for mixed-precision training [57]. Mixed-precision training is the use of lower-precision operations (float16 and bfloat16) in a model during training. This both makes it run faster and uses less memory.

4.6.2 Hardware resources

The High Performance Computing Platform IDUN [58] is used for a lot of the tasks in this thesis, especially for the training of the model. IDUN full-fills the requirements defined in Section 4.4.2.3.

Figure 4.3: Image of IDUN [59]

Chapter 5

Research Implementation and Results

This chapter presents the research implementation and results of the research questions. The chapter is divided into two parts. First, the implementation of research question 1 is described, concerning automatic smart contract code synthesis. The part of the chapter describes the implementation of research question 2, regarding generating secure smart contract code.

5.1 Implementation of RQ1

This section presents the implementation of research question 1. The implementation is done with the following steps:

1. Create verified smart contract source code dataset.
    a. Scrape verified smart contracts from the Ethereum blockchain.
    b. Normalize the smart contract files.
    c. Filter the smart contracts for uniqueness.
2. Code comment analysis.
    a. Create a parser that can parse all contract versions.
    b. Parse verified smart contract source code.
    c. Create a parsed dataset containing "comment, function" pairs.
    d. Cluster comments.
3. Language modeling
    a. Fine-tune a transformer model on the verified smart contracts dataset.

5.1.1 Data collection

5.1.1.1 Smart contract downloader

The largest provider of verified SCs is Etherscan. At `https://etherscan.io/` users can upload the source code for their deployed SC. Etherscan will then compile this source code and verify that it matches the bytecode of the deployed SC on the Ethereum blockchain. This way, other people can verify the functionality of a SC before using it. Etherscan provides a simple API for downloading verified Smart Contracts. The API is available at `https://api.etherscan.io/api`. From this endpoint, one can ask for the verified source code of a specific SC address. However, it is not guaranteed that the contract has been verified.

The following code snippet is a Google BigQuery query. It selects all SCs addresses on the Ethereum blockchain that has at least one transaction. This query was run on the 1st of April 2022, and the result was downloaded as a CSV file, available on request at `https://huggingface.co/datasets/andstor/smart_contracts/blob/main/contract_addresses.csv`. The CSV file is then used to download the SCs from Etherscan.

Code listing 5.1: Google BigQuery query for selecting all Smart Contract addresses on Ethereum that has at least one transaction.

```
SELECT contracts.address, COUNT(1) AS tx_count
FROM `bigquery-public-data.crypto_ethereum.contracts` AS contracts
JOIN `bigquery-public-data.crypto_ethereum.transactions` AS transactions
      ON (transactions.to_address = contracts.address)
GROUP BY contracts.address
ORDER BY tx_count DESC
```

The total number of files generated by the downloading program (`https://github.com/andstor/smart-contract-downloader`) was 5,810,042. In order to efficiently process these, all files were combined into a tarfile. A processing script was then created for filtering out all "empty" files. These correspond to a contract address on Ethereum that has not been verified on Etherscan.io. A total of 3,592,350 files were empty, making the source code of 38,17% of the deployed contracts on Ethereum available. Each non-empty file is then parsed and the contract data is extracted. This extraction process is rather complicated, as smart contract sources come in a wide variety of flavors and formats.

Normalization of smart contract files. SCs come in multiple flavors. To avoid confusing the machine learning model, all training data should use the same format. Hence, all contract files are normalized. The most common format is a contract written the Solidity language with a single contract entry in the file. However, a single contract file can contain multiple contracts, making use of properties like inheritance etc. The source code contracts can also be split over multiple files, a format referred to as "Multi file". When compiling these, the source code files are "flattened" into a single contract file before compilation. Another flavor is the

JSON format, which is a language that is used to describe the SCs. As can be seen in Code listing 5.2, here the source code is structured inside JSON code. Smart contracts can also be written in the Vyper language. Vyper is Pythonic programming language. Compared to Solidity, it has deliberately fewer features, making contracts more secure and easier to edit [60]. However, it is much less popular than Solidity.

Code listing 5.2: Solidity standard JSON Input format.

```
{
    "sources": {/* ... */},
    "settings": {
        "optimizer": {/* ... */},
        "evmVersion": "<VERSION>"
    }
}
```

All of the above formats are processed by the processing script, normalizing the contract source code to a single "flattened" contract file. The source code, along with the contract metadata, is then saved across multiple Parquet files, each consisting of 30000 "flattened" contracts. A total of 2,217,692 smart contracts were successfully parsed and normalized.

Filter smart contracts for uniqueness. A large quantity of Smart Contracts contains duplicated code. Primarily, this is due to the frequent use of library code, such as SafeMath [61] by OpenZeppelin [62]. Etherscan requires the library code used in a contract to be embedded in the source code. Filtering is applied to produce a dataset with a mostly unique contract source code to mitigate this. This is very important when used for model training. Failure to produce a sufficiently unique dataset would result in poor performance of a machine learning model, as it would overfit on similar data. The filtering is done by calculating the string distance between the source code. Due to the rather large amount of contracts ($\tilde{2}$ million), the comparison is only made within groups of contracts. These groups are defined by grouping on the "contract_name" for the *flattened* dataset, and by "file_name" for the *inflated* dataset. These datasets will be discussed in detail in the following sections.

The actual code filtering is done by applying a token-based similarity algorithm named Jacard Index, described in Section 2.2.3.1. The algorithm is computationally efficient and can be used to filter out SCs that are not similar to the query.

5.1.1.2 Verified Smart Contracts

The Verified Smart Contracts dataset is a dataset consisting of verified Smart Contracts from Etherscan.io. These are real SCs that are deployed to the Ethereum

Table 5.1: Verified Smart Contracts Metrics

Component	Size	Num rows	LoC
Raw	0.80 GiB	2,217,692	839,665,295
Flattened	1.16 GiB	136,969	97,529,473
Inflated	0.76 GiB	186,397	53,843,305

blockchain, containing primarily Solidity and a very small fraction Vyper code. The dataset contains multiple subsets. In the following paragraphs, these subsets are described in detail. It consists of every deployed Ethereum Smart Contract as of 1st of April 2022, whose been verified on Etherescan.io and has a least one transaction. Table 5.1 shows the metrics of the various subsets. All processing scripts are available at `https://github.com/andstor/verified-smart-contracts`. The dataset is available on request at `https://huggingface.co/datasets/andstor/verified-smart-contracts`.

Raw. The raw dataset contains mostly the raw data from Etherscan, downloaded with the smart-contract-downlader tool, as described in Section 5.1.1.1. All different contract formats (JSON, multi-file, etc.) are normalized to a flattened source code structure, as described in Section 5.1.1.1.

Flattened. The flattened dataset is a filtered version of the Raw dataset. It contains smart contracts, where every contract contains all required library code. Each "file" is marked in the source code with a comment stating the original file path: //File: path/to/file.sol. These are then filtered for uniqueness with a similarity threshold of 0.9, calculated using the Jacard index. This means that all contracts whose code shares more than 90% of the tokens will be discarded. The low uniqueness requirement is due to the often large amount of embedded library code. If the requirement is set to high, the actual contract code will be negligible compared to the library code. Most contracts will be discarded, and the resulting dataset would contain mostly unique library code. However, the dataset as a whole will have a large amount of duplicated library code. From the 2,217,692 contracts, 2,080,723 duplications are found, giving a duplication percentage of 93.82%. The resulting dataset consists of 136,969 contracts. Code listing 5.3 shows an example data instance from the dataset. The dataset is then split 80%, 10%, 10% into a training, validation and test set, respectively.

Inflated. The inflated dataset is also based on the raw dataset. Each contract file in the dataset is split into its original representative files and hence "inflated". This

Code listing 5.3: Example data instance from the flattened dataset.

```
{
  'contract_name': 'MiaKhalifaDAO',
  'contract_address': '0xb3862ca215d5ed2de22734ed001d701adf0a30b4',
  'language': 'Solidity',
  'source_code': '// File: @openzeppelin/contracts/utils/Strings.sol\r\n\r\n\r\n//
    ↪ OpenZeppelin Contracts v4.4.1 (utils/Strings.sol)\r\n\r\npragma solidity ^0
    ↪ .8.0;\r\n\r\n/**\r\n * @dev String operations.\r\n */\r\nlibrary Strings {\
    ↪ r\n...',
  'abi': '[{"inputs":[{"internalType":"uint256","name":"maxBatchSize_","type":"
    ↪ uint256"}...]',
  'compiler_version': 'v0.8.7+commit.e28d00a7',
  'optimization_used': False,
  'runs': 200,
  'constructor_arguments': '0000000000000000000000000000000000000000000000000000000000000000
    ↪ 00000000a000...',
  'evm_version': 'Default',
  'library': '',
  'license_type': 'MIT',
  'proxy': False,
  'implementation': '',
  'swarm_source': 'ipfs://e490df69bd9ca50e1831a1ac82177e826fee459b0b085a00bd7a727c8
    ↪ 0d74089'
}
```

Code listing 5.4: Example data instance from the inflated dataset.

```
{
    'contract_name': 'PinkLemonade',
    'file_path': 'PinkLemonade.sol',
    'contract_address': '0x9a5be3cc368f01a0566a613aad7183783cff7eec',
    'language': 'Solidity',
    'source_code': '/**\r\n\r\nt.me/pinklemonadecoin\r\n*/\r\n\r\n// SPDX-License-
    ↪ Identifier: MIT\r\npragma solidity ^0.8.0;\r\n\r\n\r\n/*\r\n * @dev
    ↪ Provides information about the current execution context, including the\r\n
    ↪  * sender of the transaction and its data. While these are generally
    ↪ available...',
    'abi': '[{"inputs":[],"stateMutability":"nonpayable","type":"constructor"}...]
    ↪ ',
    'compiler_version': 'v0.8.4+commit.c7e474f2',
    'optimization_used': False,
    'runs': 200,
    'constructor_arguments': '',
    'evm_version': 'Default',
    'library': '',
    'license_type': 'MIT',
    'proxy': False,
    'implementation': '',
    'swarm_source': 'ipfs://eb0ac9491a04e7a196280fd27ce355a85d79b34c7b0a83ab606d279
    ↪ 72a06050c'
}
```

mitigates a lot of the problems of the flattened dataset in terms of duplicated library code. The library code is, along with other imported contract files, split (read inflated) into separate contract records. The 2,217,692 "raw" smart contracts are inflated to a total of 5,403,136 separate contract files. These are then grouped by "file_name" and filtered for uniqueness with a similarity threshold of 0.9. This should produce a dataset with a large amount of unique source code, with low quantities of library code. A total of 5,216,739 duplications are found, giving a duplication percentage of 96.56%. The resulting dataset consists of 186,397 contracts. Code listing 5.4 shows an example data instance from the inflated dataset. The dataset is then split 80%, 10%, 10% into a training, validation and test set, respectively.

Plain text. For easy use of the dataset for casual language modeling training with HuggingFace, a "plain_text" version of both the raw, the flattened, and the inflated dataset is made available. This is done through a custom builder script

for the dataset, a feature of the Dataset library by Hugging Face. Code listing 5.5 shows an example data instance of the "plain_text" version.

Code listing 5.5: Example data instance from the plain-text version of the inflated dataset.

```
{
    'language': 'Solidity',
    'text': 'pragma solidity =0.5.16;\r\n\r\n// a library for performing overflow-
    ↪ safe math...'
}
```

5.1.2 Code comment analysis

To provide some insight into how a user can best formulate a comment for guiding the code synthesis, a cluster analysis of the comments in the smart contract dataset is conducted. First, a universal Solidity parser is constructed for parsing the Solidity code and extracting "code, comment" pairs. These results are then packaged into a dataset, and a clustering analysis is conducted. The results from this analysis are then later used in the evaluation of the code synthesis in Chapter 6 to shedd some light on which commenting style is the best to use.

5.1.2.1 Universal Solidity parser

To parse the Solidity SC, a Solidity parser is constructed. This parser has to be universally compatible with all Solidity versions, hence the grammar used needs to be a lot less restrictive than the current official Solidity grammar available from Ethereum [63]. ANTLR4 [64] is used for constructing the parser. ANTLR is a parser generator. By providing ANTLR with a formal language description called grammar, it can generate a complete parser that can automatically build parse trees. Parse trees are data structures representing how the grammar matches the input. Specifically, ANTLR4 generates a LL(*) (Left-to-right, leftmost derivation) parser [65]. ANTLR is primarily a Java application. However, several code generation targets are available, including Java, C#, Python, JavaScript, Go, C++, Swift, PHP and Dart [66]. In this project, the Python target is used.

Most programming language grammars available do not devote much effort to the handling of code comments. Comments are seen as unnecessary clutter and are normally discarded during lexing. For extracting the comments from the Solidity SC code, the original source [67] for the official Solidity grammar [63] is used. This old version is less restrictive and serves as a better starting point for ensuring support for all Solidity versions. This grammar is then simplified and made less restrictive, as well as adapted to support comments. Figure 5.1 shows a railroad diagram of a subset of the main grammar rules altered for supporting comments. The complete universal Solidity parser is made available at https://github.com/andstor/solidity-universal-parser.

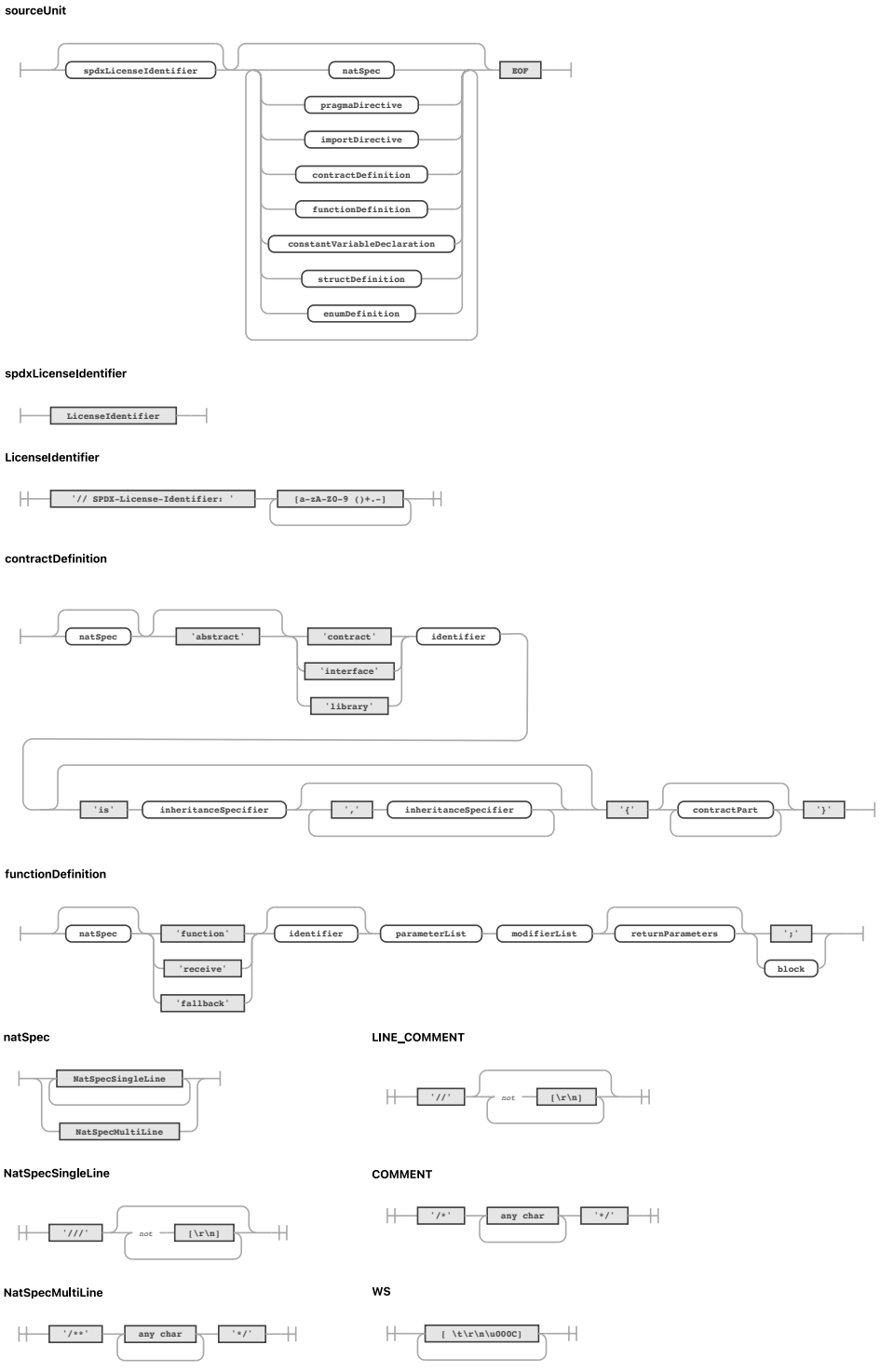


Figure 5.1: Railroad diagrams of main code comment alteration to Solidity grammar.

5.1.2.2 Verified Smart Contract Code Comments

For doing the actual extraction of the "code, comment" pairs from the inflated version of the Verified Smart Contracts dataset (see Section 5.1.1.2), the well-known visitor pattern [68] is used for visiting the parse tree generated by the universal Solidity parser. ANTLER provides basic infrastructure for implementing such a visitor. The full implementation of the visitor is available at `https://github.com/andstor/verified-smart-contracts/blob/main/script/comment_visitor.py`. A script leveraging multiprocessing is used to parallelize the parsing of the dataset. See `https://github.com/andstor/verified-smart-contracts` for instructions on how to use this script. The resulting data is then filtered for functions that do not have code comments. These are simply removed and the result is then packaged as a new dataset named Verified Smart Contract Code Comments. A total of 1.541.370 functions are extracted. Code listing 5.6 shows an example data instance from the dataset. The dataset is available on request at `https://huggingface.co/datasets/andstor/smart_contract_comments`.

5.1.2.3 Comment clustering

This section is devoted to the clustering of the comments in the parsed dataset. The comments in the dataset are first preprocessed. In contrast to normal code, code comments are of a more natural language style. Normal natural language text preprocessing is therefore employed. First, the comments are lowercased and tokenized. The default English configuration of the `word_tokenize` function from the popular Natural Language Toolkit (NLTK) [69] python library is used for tokenization. Stemming is applied to the tokenized words, using the Porter Stemmer algorithm.

For converting the tokenized comments into word embeddings, both the word2vec algorithm [70] and Term Frequency–Inverse Document Frequency (TFIDF) is used. The word2vec is able to capture some semantic similarities between the words. In particular, the implementation provided by the gensim library [71] is used. The algorithm is configured to produce 100-dimensional vectors, using a window size of 5, and a minimum count of 5.

To weed out the most frequent words TFIDF is also applied. For example, the different commenting types all start each line with a special word, such as "//", "///" or "*". By using TFIDF, it is possible to get more insights into the different ways of writing comments, beyond just the formatting style of the comments. The resulting word embeddings from the word2vec and TFIDF are multiplied. For each comment, the resulting word embeddings are averaged to form a final comment (or document) embedding.

The comment embeddings are clustered using the K-means algorithm. The number of clusters k is determined by using the Elbow method for deciding the optimal number of clusters. The results from the Elbow method are presented in Figure 5.2. From the curve, it is not entirely obvious where the "elbow" is. However, a k of 4 is selected. For visually inspecting the clustered comments result, the

Code listing 5.6: Example data instance from the inflated dataset.

```
{
    'contract_name': 'BondedECDSAKeep',
    'file_path': '@keep-network/keep-core/contracts/StakeDelegatable.sol',
    'contract_address': '0x61935dc4ffc5c5f1d141ac060c0eef04a792d8ee',
    'language': 'Solidity',
    'class_name': 'StakeDelegatable',
    'class_code': 'contract StakeDelegatable {\n    using OperatorParams for uint25
    ↪ 6;\n\n    mapping(address => Operator) internal operators;\n\n    struct
    ↪ Operator {\n        uint256 packedParams;\n        address owner;\n
    ↪ address payable beneficiary;\n        address authorizer;\n    }\n\n...',
    'class_documentation': '/// @title Stake Delegatable\n/// @notice A base
    ↪ contract to allow stake delegation for staking contracts.',
    'class_documentation_type': 'NatSpecSingleLine',
    'func_name': 'balanceOf',
    'func_code': 'function balanceOf(address _address) public view returns (uint256
    ↪  balance) {\n        return operators[_address].packedParams.getAmount();\n
    ↪     }',
    'func_documentation': '/// @notice Gets the stake balance of the specified
    ↪ address.\n/// @param _address The address to query the balance of.\n///
    ↪ @return An uint256 representing the amount staked by the passed address.',
    'func_documentation_type': 'NatSpecSingleLine',
    'compiler_version': 'v0.5.17+commit.d19bba13',
    'license_type': 'MIT',
    'swarm_source': 'bzzr://63a152bdeccda501f3e5b77f97918c5500bb7ae07637beba7fae76
    ↪ dbe818bda4'
}
```

100-dimensional vectors are reduced to 2D using Principal Component Analysis (PCA). The explained variance captured in the 2D plot is approximately 0.64, as shown in the Scree Plot in Figure 5.3. The clustering result is shown in Figure 5.4.

Code listings 5.7 to 5.10 shows an example from each of the four clusters. Upon manual inspection of the different clusters, several patterns emerge. Cluster 0 is mainly composed of comments that is almost exclusively made up of NatSpec comments with only NatSpec fields, for example the "@parameter" and "@return" fields. Most comments also start with a brief description of the function, as for example line 1 in Code listing 5.7. Next, cluster 1 consists of one-liners, briefly describing what the function. Cluster 2 contains more lengthy comments that describe the function in detail. It is similar to cluster 1 but does not make significant use of the NatSpec fields. Several of these comments are from some implementation of common libraries. For example, Code listing 5.9 shows a comment for the implementation of a transfer function in a contract implementation of a ERC20 token. Compared to the base implementation by the OpenZeppelin library [62], this version adds 1.7% tax if the sender or recipient is an exchange (lines 8-10). Finally, cluster 3 contains that presents a more "artistic" nature. For example, Code listing 5.10 marks the start and end of the comments with many dashes.

Code listing 5.7: NatSpec single-line comment in cluster 0.

```
1  /// @dev Executes the next transaction only if the cooldown has passed and the
       transaction has not expired
2  /// @param to Destination address of module transaction
3  /// @param value Ether value of module transaction
4  /// @param data Data payload of module transaction
5  /// @param operation Operation type of module transaction
6  /// @notice The txIndex used by this function is always 0
```

Code listing 5.8: Single-line comment in cluster 1.

```
1  // Allow the owner to cash out the holdings of this contract.
```

Code listing 5.9: NatSpec multi-line comment in cluster 2.

```
1  /**
2   * @dev See {IERC20-transfer}.
3   *
4   * Requirements:
5   *
6   * - ‘recipient‘ cannot be the zero address.
7   * - the caller must have a balance of at least ‘amount‘.
8   *
9   * If recipient or sender is exchange, transaction will be taxed 1.7%
10  * Tax is sent to our taxAddress
11  */
```

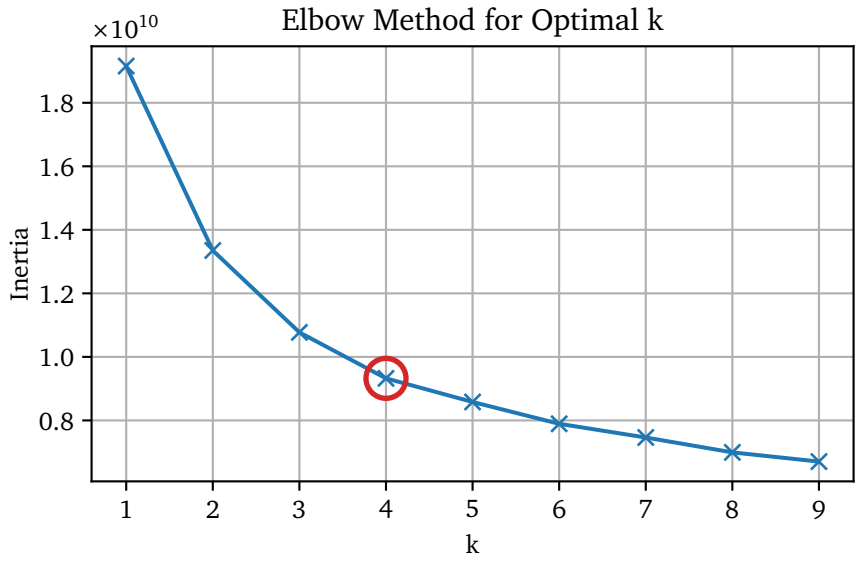


Figure 5.2: Elbow method for determining the optimal number of clusters.

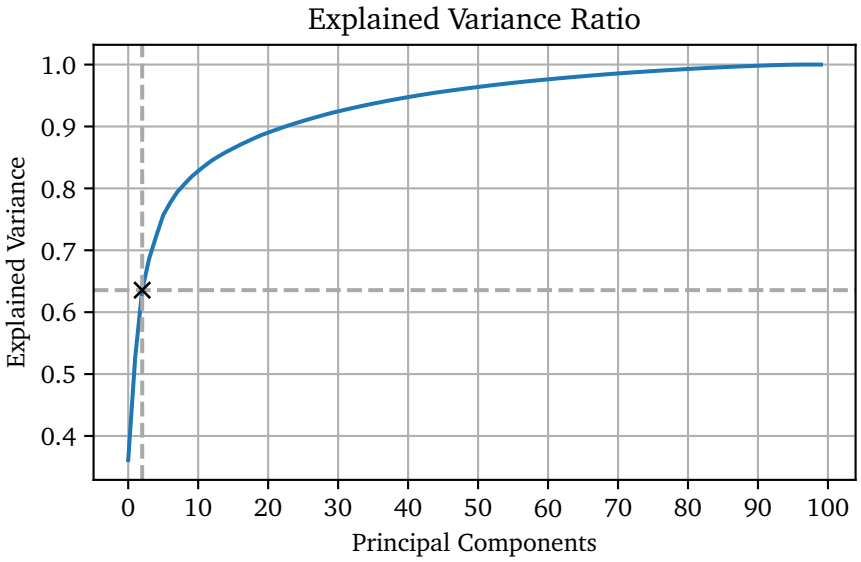


Figure 5.3: Scree Plot for the PCA dimensionality reduction

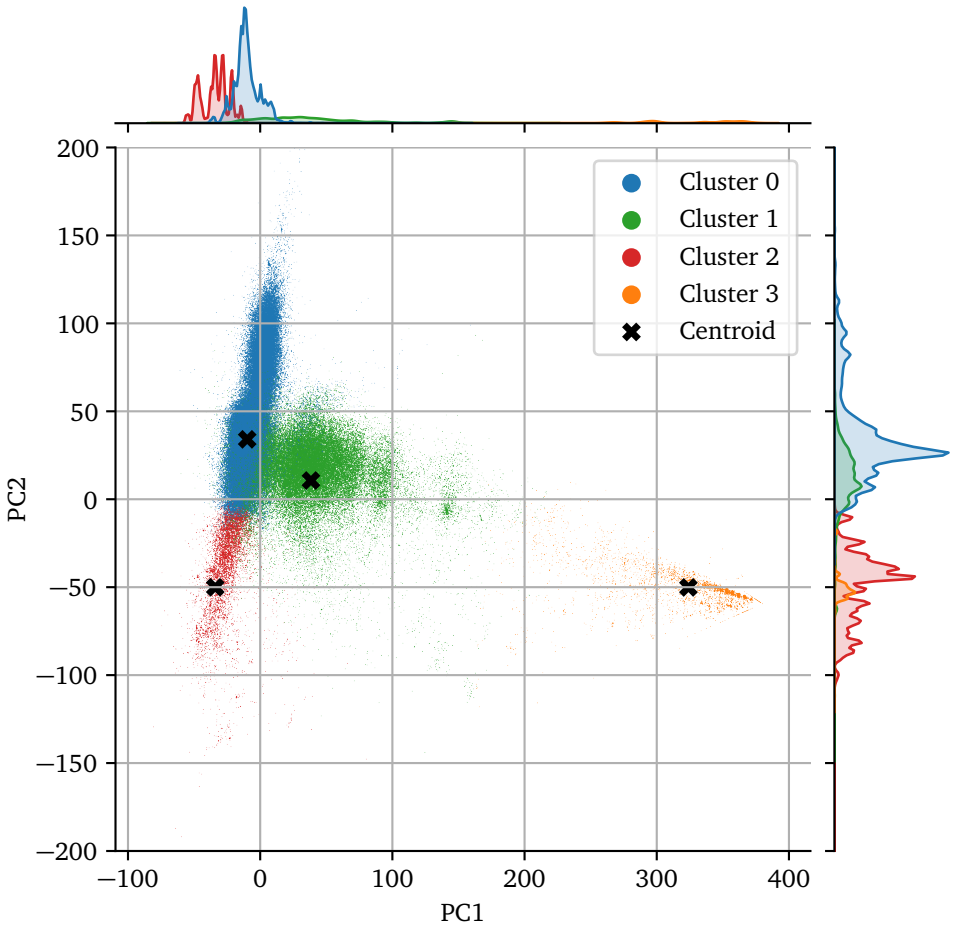


Figure 5.4: 2D plot of the comment clusters.

Code listing 5.10: Custom comment style from cluster 3

```
// ----------------------------------------------------------------------
// Returns the amount of tokens approved by the owner that can be
// transferred to the spender's account
//
// THIS TOKENS ARE NOT TRANSFERRABLE.
//
// ----------------------------------------------------------------------
```

Table 5.3: GPT-J-6B model details.

Hyperparameter	**Value**
n_parameters	6,053,381,344
n_layers	28
d_model	4,096
d_ff	16,384
n_heads	16
d_head	256
n_ctx	2,048
n_vocab	50,257 (same tokenizer as GPT-2/3)
position & encoding	Rotary Position Embeddings (RoPEs)
RoPE dimensions	64

5.1.3 Language Modeling

This section presents a detailed overview of the system architecture for generating Smart Contract code. The first section describes the specific configuration of the pre-trained model used. Following is a section that describes the fine-tuning process on the inflated Verified Smart Contract dataset presented in Section 5.1.1.2.

5.1.3.1 Pre-training

In this project, pre-trained weights for GPT-J-6B from EleutherAI are used. See Section 4.4.2.2 for a description of the model architecture. The pre-training by EleutherAI is done on the dataset The Pile, described in Section 4.4.2.1. Of the roughly 825GiB, 95.16 GiB (7.59%) of The Pile is code from GitHub. Compared to many other open-source models, GPT-J-6B is one of the most promising models for the task of code generation.

The specific GPT-J model configuration can be seen in Table 5.3. In detail, GPT-J-6B consists of 28 layers with a model dimension of 4096, and a feedforward dimension of 16384. The model dimension is split into 16 heads, each with a dimension of 256. Rotary Position Embedding (RoPE) is applied to 64 dimensions of each head. The model is trained with a tokenization vocabulary of 50257, using the same set of Byte-Pair Encodings (BPEs) as GPT-2 and GPT-3. The weights of GPT-J-6B are licensed under version 2.0 of the Apache License. When assessed on the validation split of the inflated Verified Smart Contract dataset Section 5.1.1.2, the model achieves an accuracy of 0.800 and a perplexity of 2.600.

5.1.3.2 Fine-tuning

To improve the pre-trained GPT-J-6B model's smart contract code generation performance, the model is fine-tuned on a dataset only containing real Ethereum Smart Contract code. Specifically, the model is fine-tuned on the training split of the plain-text Section 5.1.1.2 version of the inflated Verified Smart Contracts dataset Section 5.1.1.2. The fine-tuning task used is the same as for the pre-training task, namely Casual Language Modeling (CLM). The model is fed a complete SC all at once, and then internal masking is applied to prevent the model from cheating by looking at future tokens. For more details on the inner workings of the training procedure, see Section 2.1.2. Before training, the dataset is randomly shuffled. For running the training process, the CLM script [1] provided by HuggingFace is used.

Due to the huge size of the GPT-J-6B model, the deep learning optimization library DeepSpeed [50] is used as a wrapper around the HuggingFace library. See Section 4.6.1 for more details of the DeepSpeed library. While DeepSpeed enables the training of virtually arbitrary-sized models, there is a tradeoff between model size and training speed. In this project, several DeepSpeed configurations were tried out to successfully load and train the model without encountering an Out of Memory (OOM) error, while still maintaining adequate training speed. Using ZeRO-2 with CPU offloading (ZeRO-Offload), mixed-precision (bfloat16), a batch size of 1, and 16 gradient accumulation steps, it is possible to load and efficiently train the model using 10 x NVIDIA A100 GPUs with 40GB memory [2]. The computing node used for training has 48 CPUs available, along with 1.47 terabytes of RAM. Figure 5.5 shows a screenshot of the nvidia-smi program during the training of the model. As can be seen from the figure, all GPUs are at 100% utilization. Figure 5.6 presents a screenshot of the htop program showing host CPU and memory activity during optimizer computation. The command for running the HuggingFace training script while using DeepSpeed is shown in Code listing 5.11. A complete list of the hyperparameters used for training the model is available in Table 5.5, along with the DeepSpeed configuration in Table 5.7. All training scripts and configurations used are available at `https://github.com/andstor/smart-contract-code-generation`.

The training process is run for two epochs. At every 5 steps, the model is evaluated on 256 samples from the validation split of the Verified Smart Contracts dataset. Figure 5.7 shows a graph over the training and evaluation loss during training. Figure 5.8 shows a graph over the evaluation accuracy during training. The training is completed after 7 days and 4s hours. After completion of the training, the model is evaluated on the entire validation split, achieving a total accuracy of 0.917 and perplexity of 1.510. The fine-tuned model is available on request at `https://huggingface.co/andstor/gpt-j-6B-smart-contract`.

[1] `https://github.com/huggingface/transformers/blob/v4.19.0/examples/pytorch/language-modeling/run_clm.py`

[2] `https://www.nvidia.com/en-us/data-center/a100/`

Table 5.5: Hyperparameters for GPT-J model

Hyperparameter	
learning_rate	5e-05
train_batch_size	1
eval_batch_size	1
seed	42
distributed_type	multi-GPU
num_devices	10
gradient_accumulation_steps	16
total_train_batch_size	160
total_eval_batch_size	10
optimizer	Adam with betas=(0.9,0.999) and epsilon=1e-08
lr_scheduler_type	linear
num_epochs	2.0

Code listing 5.11: Command for running the HuggingFace CLM training script with DeepSpeed.

```
deepspeed --hostfile=hostfile run_clm.py \
    --deepspeed ds_zero2_bf16.json \
    --model_name_or_path EleutherAI/gpt-j-6B \
    --dataset_name andstor/smart_contracts \
    --dataset_config_name plain_text \
    --output_dir ./out \
    --report_to wandb \
    --validation_split_percentage 20 \
    --save_steps 250 \
    --do_train --do_eval \
    --logging_first_step --logging_steps 1 \
    --num_train_epochs 2 \
    --evaluation_strategy steps --eval_steps 5 \
    --max_eval_samples 256 \
    --block_size 1024 \
    --bf16 \
    --gradient_accumulation_steps 16 --eval_accumulation_steps 16 \
    --per_device_train_batch_size 1 --per_device_eval_batch_size 1
```

Table 5.7: DeepSpeed Zero configuration.

Hyperparameter	
stage	2
contiguous_gradients	true
reduce_scatter	true
reduce_bucket_size	2.000000e+08
allgather_partitions	true
allgather_bucket_size	2.000000e+08
overlap_comm	true
load_from_fp32_weights	true
elastic_checkpoint	false
cpu_offload	true
sub_group_size	1.000000e+09
prefetch_bucket_size	5.000000e+07
param_persistence_threshold	1.000000e+05
max_live_parameters	1.000000e+09
max_reuse_distance	1.000000e+09
gather_16bit_weights_on_model_save	false
ignore_unused_parameters	true
round_robin_gradients	false
legacy_stage1	false

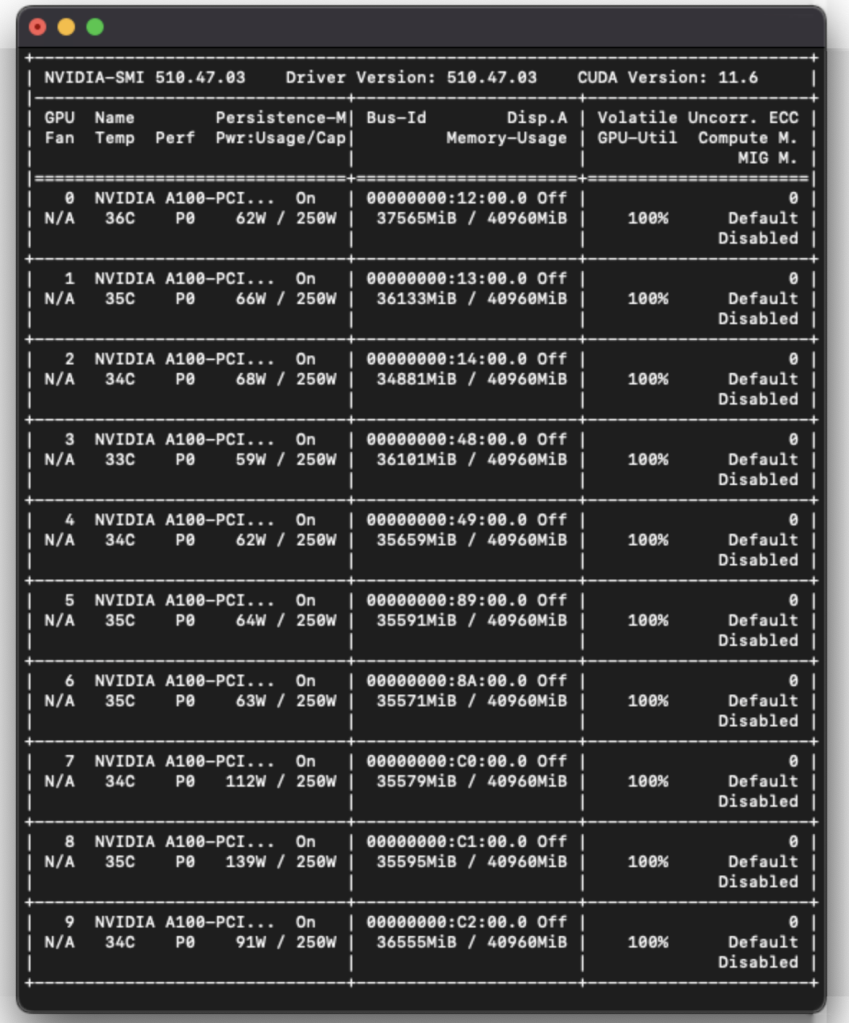


Figure 5.5: Screenshot of nvidia-smi program showing 100% GPU utilization.

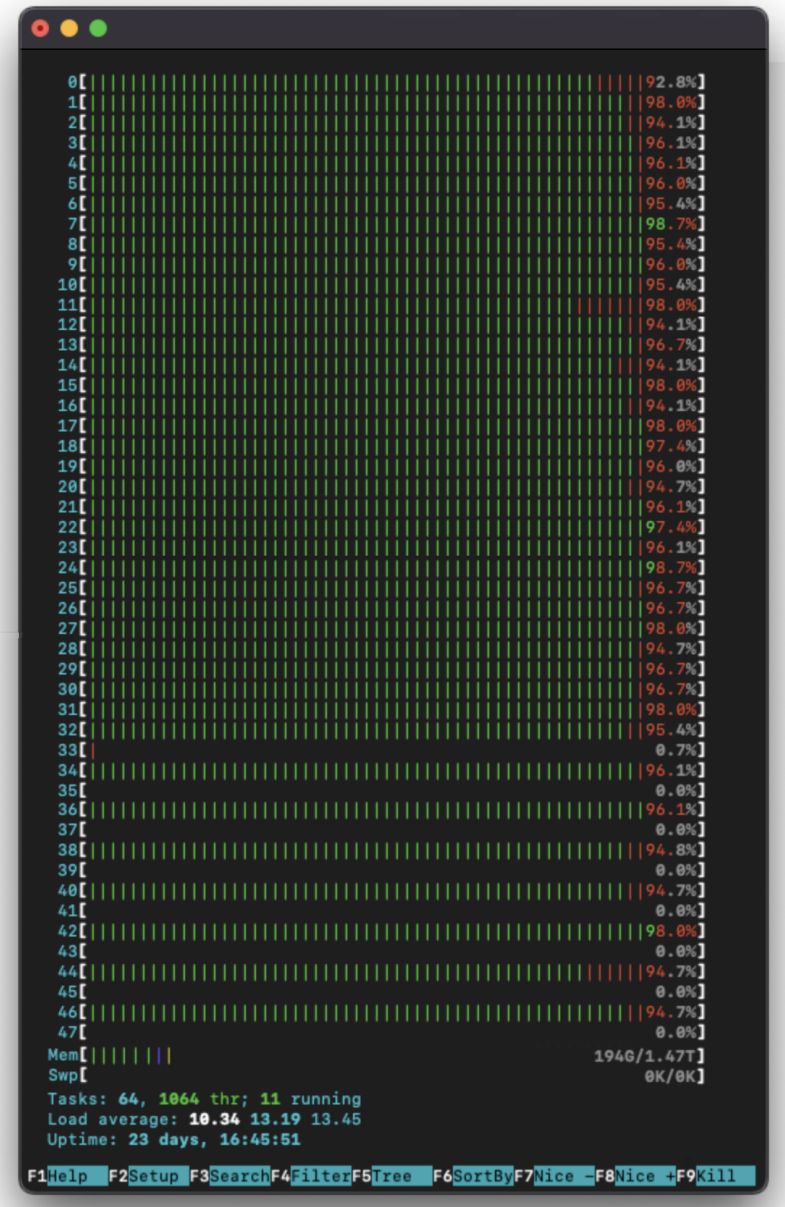


Figure 5.6: Screenshot of htop program showing host CPU and memory activity during optimizer computation.

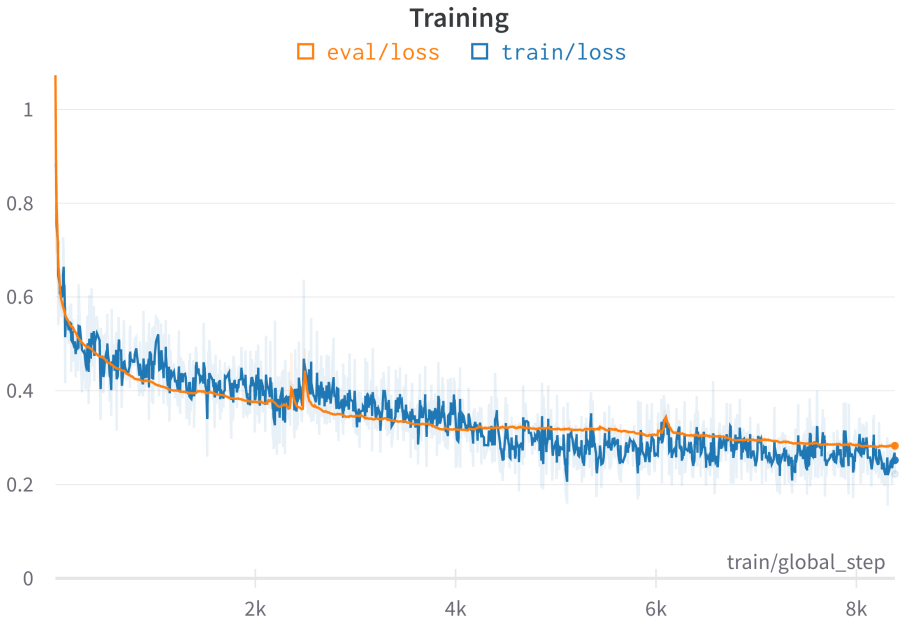


Figure 5.7: Training and evaluation loss during model training.

Figure 5.8: Evaluation accuracy during model training.

5.2 Implementation of RQ2

This section presents the implementation of research question 2. Primarily, this section describes the implementation of the new technique named security conditioning, described in Section 4.5.1. In security conditioning, the training data is augmented with security labels, stating either secure or vulnerable. The main alteration needed is in the training data. The implementation is done with the following steps:

1. Create an audited version of the smart contract dataset
    a. Label the smart contracts with a vulnerability detection tool.
2. Language modeling
    a. Fine-tune a transformer model on the audited verified smart contract dataset, employing security conditioning.

5.2.1 Data preparation

5.2.1.1 Vulnerability labeling

For labeling the SCs as vulnerable or secure, the Java program SolDetector by Tianyuan Hu [72] is used. The choice of using SolDetector is due for two reasons. Firstly, as SolDetector is ontology-based, it does not need a complete contract file with all code dependencies. This makes it possible to use the inflated dataset version (see Section 5.1.1.2). Other vulnerability detection tools such as (Oyente) [73] that for example use symbolic analysis, would only work on the flattened dataset version (see Section 5.1.1.2). SolDetector works on both. Secondly, SolDetector works with any Solidity version.

SolDetector takes in a SC file and outputs the vulnerability analysis results as a text file. In this text file, the detected vulnerability type(s) and the offending line(s) are reported. The paper [72] of SolDetector also classifies the different vulnerabilities according to risk level. Due to the large number of contracts needed to be labeled in this project, SolDetector is run in parallel. A python script is created that leverages multiprocessing to run SolDetector in parallel. Since SolDetector is a Java program, it is run as a child process and controlled with the help of the python module Pexpect [74]. Since starting and stopping Java applications are time-consuming, extra care is taken to ensure that each instantiated Solidity process is kept alive for as long as possible and only restarted when necessary. Figure 5.9 shows a screenshot from running the processing script using 40 processes. The vulnerability processing scripts are available at `https://github.com/andstor/verified-smart-contracts-audit`.

5.2.1.2 Verified Smart Contracts Audit dataset

The results of the vulnerability labeling are packed as subsets into a dataset named Verified Smart Contracts Audit. This is done for both the flattened and inflated

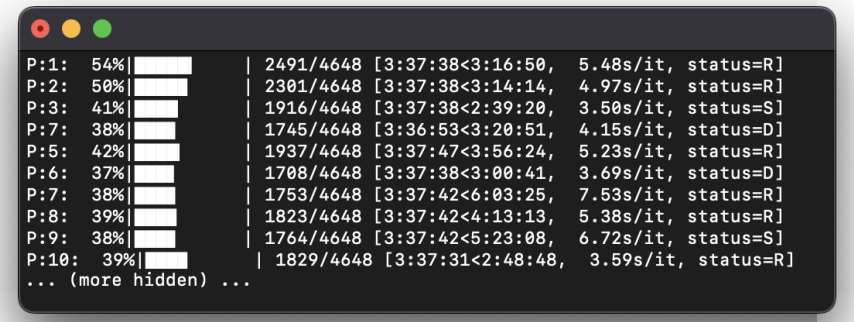


Figure 5.9: Screenshot from the vulnerability labeling process with SolDetector.

dataset versions. The finished dataset is available on request at `https://huggingface.co/datasets/andstor/smart_contracts_audit`. Both dataset versions keep the original split into a training, validation and test set (80%, 10%, 10%). Code listing 5.12 shows an example data instance from the audited inflated dataset.

Code listing 5.12: Example data instance from the audited inflated dataset.

```
{
    'contract_name': 'OceanWorld',
    'file_path': 'OceanWorld.sol',
    'contract_address': '0xe19c5ea08f26af53bf7da7da5e727bb2c5c69f95',
    'language': 'Solidity',
    'source_code': 'pragma solidity ^0.8.0; contract OceanWorld is ERC721Enumerable
    ↪ ...',
    'defects': '[{"defect": "Nest_Call", "type": "Business_logic", "severity": "
    ↪ High", "lines": ["193", "125", "165"]}, {"defect": "Frozen_Ether", "type":
    ↪ "Code_specification", "severity": "Medium", "lines": ["3"]}, {"defect": "
    ↪ Exceed_authority_access", "type": "Access_control", "severity": "Medium", "
    ↪ lines": ["31"]}]',
    'compiler_version': 'v0.8.7+commit.e28d00a7',
    'license_type': 'MIT',
    'swarm_source': 'ipfs://36f4cbcbeca01a804a52ae73931c970301e46d79022cdf26e6e6158
    ↪ d9105fe83'
}
```

Figure 5.10 shows a doughnut chart over the distribution of the vulnerability severities in the flattened dataset at different granularity levels, where each level occurs at least once in the SC. The outer ring shows the additional security levels for each contract. For example, "HML" means that the contract has at least

three vulnerabilities with the corresponding "High", "Medium", and "Low" security levels. As can be seen in the figure, almost three-quarters of the contracts contain at least one high-risk vulnerability. Figure 5.11 shows the distribution of the different types of vulnerabilities in the flattened dataset, categorized by severity level. Notably, a significant portion of the high-severity vulnerabilities is integer overflow and underflow vulnerabilities. Figures 5.12 and 5.13 presents the same vulnerability distribution chart for the audited contracts in the inflated dataset. The distribution of vulnerability types follows the same characteristics as for the flattened dataset. However, only around half of the contracts contain at least one high-risk vulnerability. As described in Section 5.1.1.2, the main intention behind the inflated dataset is to reduce the amount of library. Hence, one can deduce that a significant portion of the vulnerabilities come from various SC libraries.

Embedded. For easy use of the labeled dataset for Casual Language Modeling (CLM) training, an "embedded" version of both the flattened and the inflated dataset is made available. This is done through a custom builder script for the dataset, a feature of the Dataset library by Hugging Face. The builder script parses the contract audit and determines whether the contract is secure or vulnerable. Based on this analysis, it then prepends "<|secure|>" or "<|vulnerable|>" to the top of the contract source code. In this project, a contract is considered secure if it does not contain any high-risk vulnerabilities. Otherwise, the contract is considered vulnerable. This also makes the inflated dataset balanced, as about 50% of the contracts are secure and 50% are vulnerable (see Figure 5.12).

5.2.2 Language Modeling

This section presents the language modeling procedure using the security conditioning technique proposed in Section 4.5.1 for generating secure Smart Contract code. In security conditioning, the training data is augmented with security labels, stating either secure or vulnerable. This data augmentation is implemented by the embedded version of the Verified Smart Contracts Audit dataset Section 5.2.1.2, by adding "<|secure|>" or "<|vulnerable|>" to secure or vulnerable contracts. To make the most use of the security labels, a small alteration to the tokenizer is made, as described in the following section. Otherwise, the language modeling procedure is more or less identical to the one used for RQ1 Section 5.1.3.

5.2.2.1 Tokenizer

Depending on the security labels and the type of tokenizer used, the tokenizer might decide to split the security label into multiple, already pre-trained, tokens. For example, the "<|secure|>" label is tokenized into five different tokens: ’<’, ’|’, ’secure’, ’|’, ’>’ with corresponding ids: 27, 91, 22390, 91, 29. These tokens might also be part of making up other words. This might confuse the model during training, making it harder for it to successfully condition on the labels. To mitigate

Figure 5.10: Doughnut chart over the distribution of the vulnerability severities in the flattened dataset at different granularity levels, where each level occurs at least once in the SC.

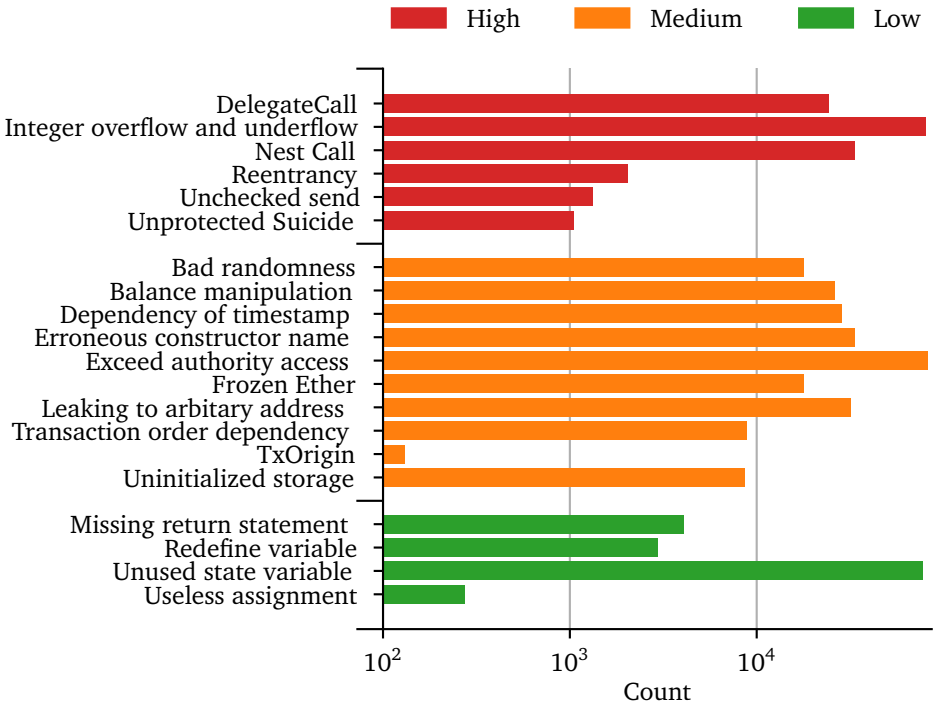


Figure 5.11: Distribution of vulnerabilities in the flattened dataset.

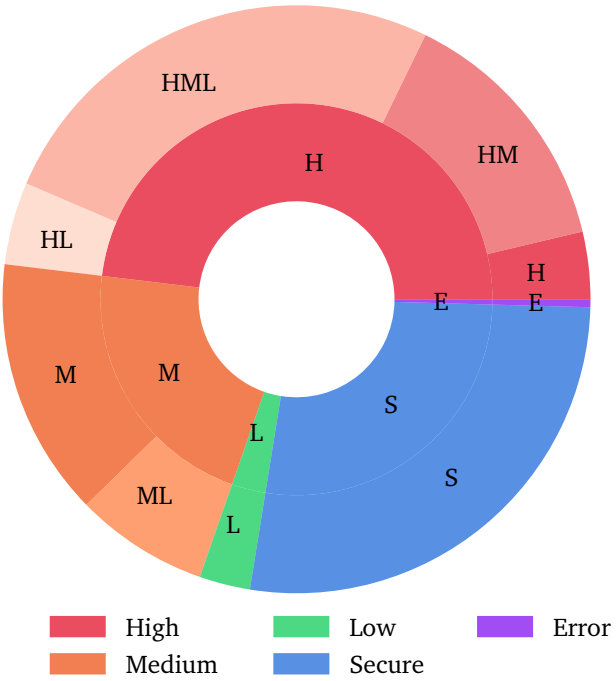


Figure 5.12: Doughnut chart over the distribution of the vulnerability severities in the inflated dataset at different granularity levels, where each level occurs at least once in the SC.

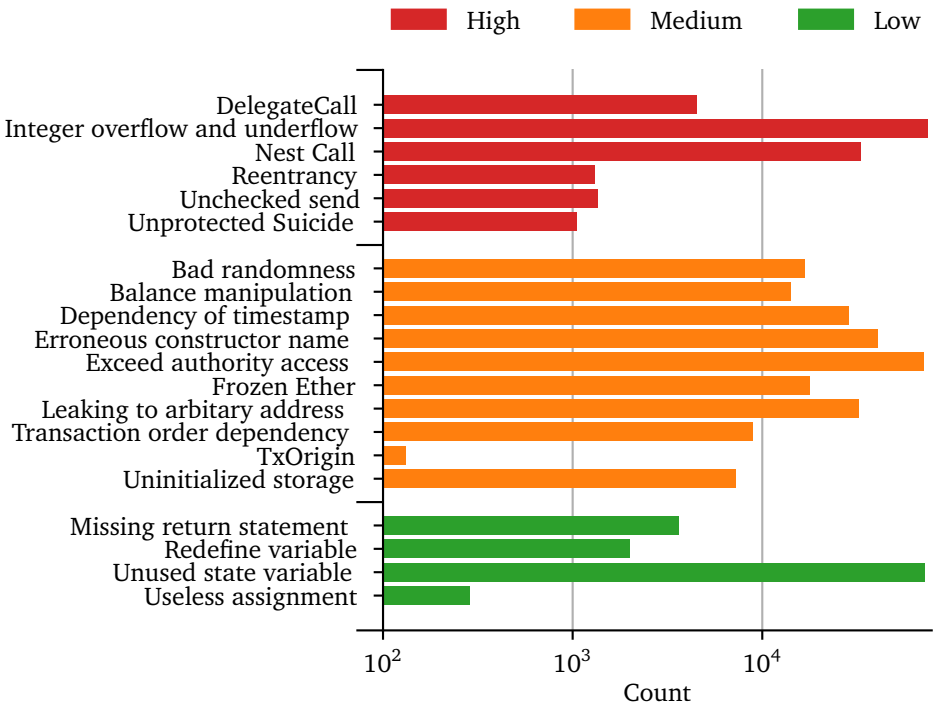


Figure 5.13: Distribution of vulnerabilities in the inflated dataset.

this, the security labels "<|secure|>" and "<|vulnerable|>" are added as special tokens to the tokenizer, effectively expanding the vocabulary. The "<|secure|>" label is now instead tokenized as "<|secure|>" with id 50400. This change also requires resizing the model's embedding matrix. The two added embeddings are randomly initialized.

5.2.2.2 Fine-tuning

For fine-tuning the model on the embedded version of the Verified Smart Contracts Audit dataset Section 5.2.1.2, the same procedure and hyperparameters as in RQ1 are used. The training process is run for two epochs. At every 5 steps, the model is evaluated on 256 samples from the validation split of the Verified Smart Contracts Audit dataset. Figure 5.14 shows a graph of the training and evaluation loss during training. Figure 5.15 shows a graph over the evaluation accuracy during training. The training is completed after 7 days and 4 hours. After completion of the training, the model is evaluated on the entire validation split, achieving a total accuracy of 0.917 and perplexity of 1.510. Compared to the fine-tuned model without security conditioning (see Section 5.2.2.2), the technique does not introduce any significant performance decrease in terms of neither accuracy nor perplexity. The fine-tuned model is available on request at `https://huggingface.co/andstor/gpt-j-6B-smart-contract-audit`.

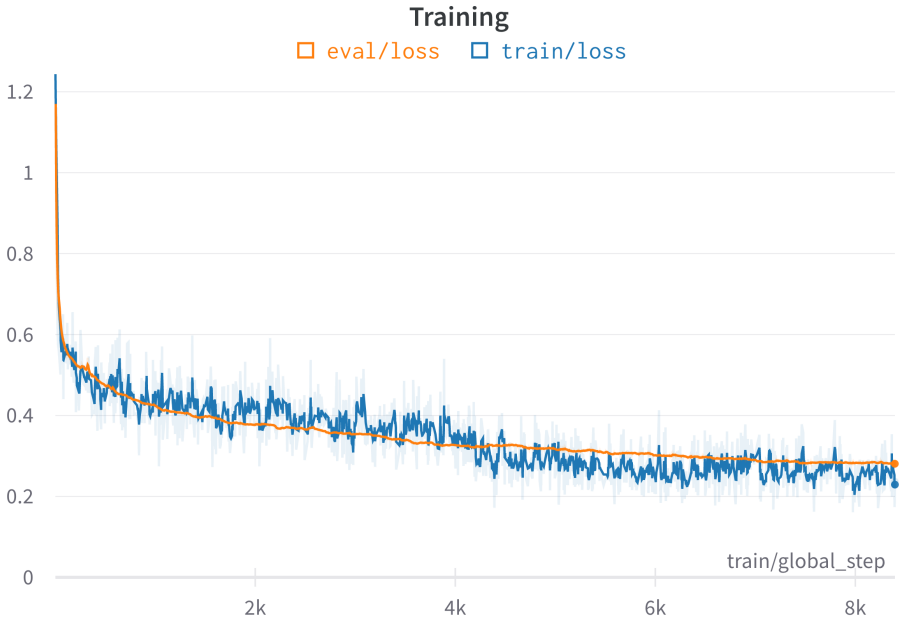


Figure 5.14: Training and evaluation loss during training of model with security conditioning.

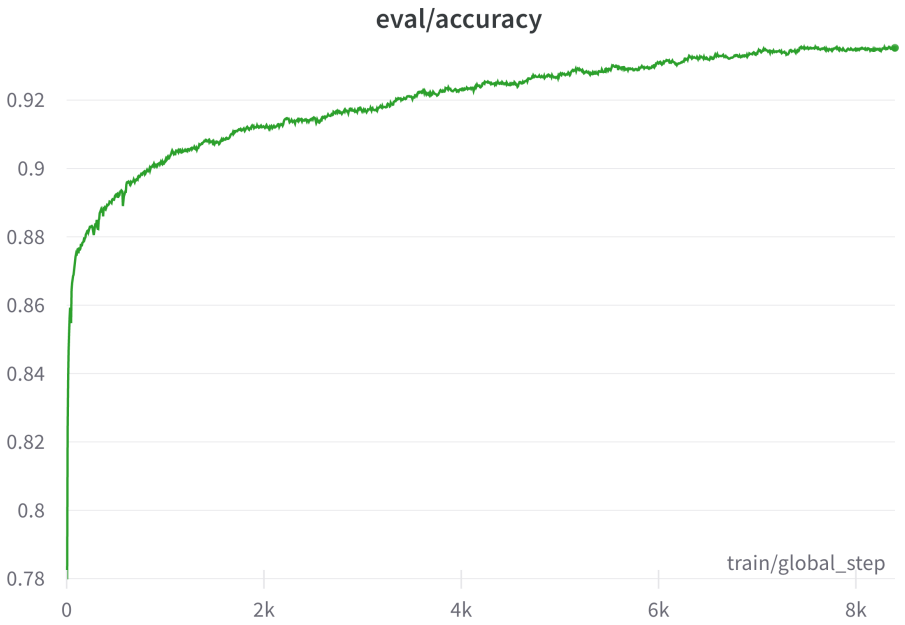


Figure 5.15: Evaluation plot of accuracy during training of model with security conditioning.

Chapter 6

Evaluation

This chapter presents the evaluation of the research questions. First, RQ1 is presented in Section 6.1. The evaluation of RQ2 is presented in Section 6.2.

6.1 Evaluation of RQ1

This section evaluates the performance of the implementation developed for research question 1. First, the evaluation method is presented, followed by a description of the metrics used. Finally, the evaluation results are presented. We evaluate two scenarios. In the first scenario, only comments are used as input to the model. Then, the model is evaluated using a comment-aided approach, using both comments and code as input.

6.1.1 Evaluation Method

The evaluation strategy employed is to measure the similarity between generated code and original code. This evaluation strategy captures how such a system would perform in real life. The conceptual evaluation strategy consists of four steps:

1. Some code from a real SC is extracted.
2. The extract is split into two parts.
3. The first part is fed as input to the model, while the second part (original code) is used as the target value.
4. The generated output is then compared to the target value to calculate their similarities.

As RQ1 is concerned with the use of a comment-aided approach for generating code, all evaluation runs include the use of comments as the primary input. Hence, the split is done between a function and its comment. However, the amount of context (supporting code) is varied. Code listing 6.1 demonstrates how the different parts of a contract are used in the evaluation. Lines 1-11 are used as the code context, while lines 13-14 are the comment. The target code is in lines

15-20. All consecutive lines of code are discarded. In Code listing 6.1, this would only be line 21. First, an evaluation run is done in Section 6.1.3 using the normal way of generating functions from comments, meaning only comments are used as input. Section 6.1.4 runs an evaluation utilizing all the available code context, the core of the comment-aided approach. Since the model is auto-regressive, a custom stopping strategy based on matching braces is implemented for generating well-formed functions. Further, all code generations use the default *temperature* of 1.

Code listing 6.1: Different contract parts.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >= 0.7.0;
3
4  contract Coin {
5      // Sends an amount of newly created coins to an address
6      // Can only be called by the contract creator
7      function mint(address receiver, uint amount) public {
8          require(msg.sender == minter);
9          require(amount < 1e60);
10         balances[receiver] += amount;
11     }
13     // Sends an amount of existing coins
14     // from any caller to an address
15     function send(address receiver, uint amount) public {
16         require(amount <= balances[msg.sender], "Insufficient balance.");
17         balances[msg.sender] -= amount;
18         balances[receiver] += amount;
19         emit Sent(msg.sender, receiver, amount);
20     }
15 }
```

6.1.2 Evaluation metrics

For comparing the generated code to the original code as described in Section 6.1.1, the BLEU score is used. The metric is described in detail in Section 2.2.2.1. BLEU is a commonly used evaluation metric within the area of code synthesis [75]. However, as the metric was originally designed for evaluating natural language, it does have its shortcomings when applied for automatic evaluation of code synthesis. In particular, it neglects important syntactic and semantic features of codes [75]. Because of this, adaptations such as CodeBLEU by Ren *et al.* [75] have emerged, incorporating ASTs and data-flow analysis. However, there is currently no readily available implementation, especially for SC code. Recent works such as [2, 3, 76]

have turned to using functional correctness for evaluation, where the generated code is evaluated by unit testing. However, this approach requires the curation of testing datasets, such as the hand-written Python evaluation set HumanEval [1]. Further, unit testing Solidity code is not a straightforward approach, as it normally involves the EVM. Because of this, this project settles with using BLEU score for evaluation and leaves alternative evaluation methods for SC code synthesis for future research.

6.1.3 Comment only evaluation

To provide some insights into how to best formulate the comments for the model, an evaluation run is done using only comments as input, without any supporting code context. First, the testing split of the Verified Smart Contract Code Comments dataset is filtered according to the four clusters identified in Section 5.1.2.3. From each of these clusters, a total of 10.000 random samples are drawn. However, only 4000 samples from cluster 3 (zero-indexed) were available in the testing split. From the samples, only the function comment is fed into the model as input. The function from the sample is then compared to the generated function by calculating the BLEU score. This evaluation procedure is done for both the pre-trained model and the fine-tuned model.

Figure 6.1 shows a density histogram of the BLEU score results of the evaluation. The left column of plots shows the BLEU score distribution of the pre-trained model, while the right column shows the BLEU score distribution of the fine-tuned model. The first row of plots shows the results from cluster 0, the second row from cluster 1, the third row from cluster 2, and the fourth row from cluster 3. Generating function code using only comments is an exceptionally hard task as the search area for a potential solution is extremely large. It is therefore expected to see a lot of BLEU scores of 0. Indeed, this is the case in all of the plots. However, from the plots, it is clear that the fine-tuned model performs significantly better than the pre-trained model, as the pre-trained model is almost not able to produce any scores above 0.025. The averaged BLEU scores of each cluster can be seen in Table 6.1.

The performance for the fine-tuned model is ranked from worst to best as follows: cluster 1, cluster 0, cluster 3 and cluster 2. For the fine-tuned model, Cluster 0 and cluster 1 show similar distribution characteristics, with Cluster 1 performing a bit better. A large part of the comments in cluster 0 is devoted to function parameter descriptions (see Code listing 5.7). These parameter description results in about a 55% increase of the BLEU score. However, both cluster 2 and cluster 3 significantly outperform cluster 0, as can be seen from Table 6.1. Cluster 2 is the best performing of all the clusters. As discovered in Section 5.1.2.3, this cluster contains a lot of library code implementations. Therefore, it is reasonable to assume that the model excels in generating code for the implementation of popular libraries, as it might have memorized parts of these libraries. Cluster 3 also

[1] `https://github.com/openai/human-eval`

Table 6.1: Average BLEU score of only comment generation.

Pre-trained model	Cluster 0	Cluster 1	Cluster 2	Cluster 3
Pre-trained model	0.065	0.002	0.009	0.019
Fine-tuned model	0.282	0.167	0.456	0.397

performs rather well. However, it presents a rather interesting distribution with some large peaks to the far right. Upon manual inspection, it is clear that most of these spikes are part of a popular ERC20 [2] token implementation from an old tutorial [77] from 2017. This is also the case for the pre-training plot. As there are multiple forks of this code available on GitHub, it is most likely included in The Pile (see Section 4.4.2.1).

6.1.4 Comment + code context evaluation

Normally, during the inference of transformer models, the longer the input sequence (context) - the better the performance. For evaluating the "optimal-case" performance of the model, an evaluation run is done by providing extensive code context to the input. This is a typical use-case scenario of the system, where a user already has written some code and wants to extend it. The user can then simply write a new comment describing the desired new functionality, and then ask the model to suggest some automatically generated code, using everything the user has typed so far as input.

A total of 10.000 random samples are drawn from the test split of the Verified Smart Contract Code Comments dataset. Each drawn sample contains function "code, comment" pairs, as well as the complete contract code from which the function was extracted. The original contract code is then cut at the end of the sampled function comment. This is then fed into the model as input to generate code, and the BLEU score is calculated by comparing the generated code against the actual function. This evaluation procedure is done for both the pre-trained model and the fine-tuned model.

Figures 6.2 and 6.3 shows a histogram of the BLEU score results of the evaluation. Comparing the two figures, it is clear that the fine-tuned model performs much better than the pre-trained model. The distribution of the BLEU scores of the pre-trined model (Figure 6.2) shows two interesting characteristics. First, almost half of the 10.000 samples achieve a BLEU score close to 0. This means that the generated output is completely different from the target code. Second, the rest of the histogram presents a rather uniform distribution of low BLEU scores. Hence, the pre-trained model does not perform well for generating SCs. The results from

[2] `https://eips.ethereum.org/EIPS/eip-20`

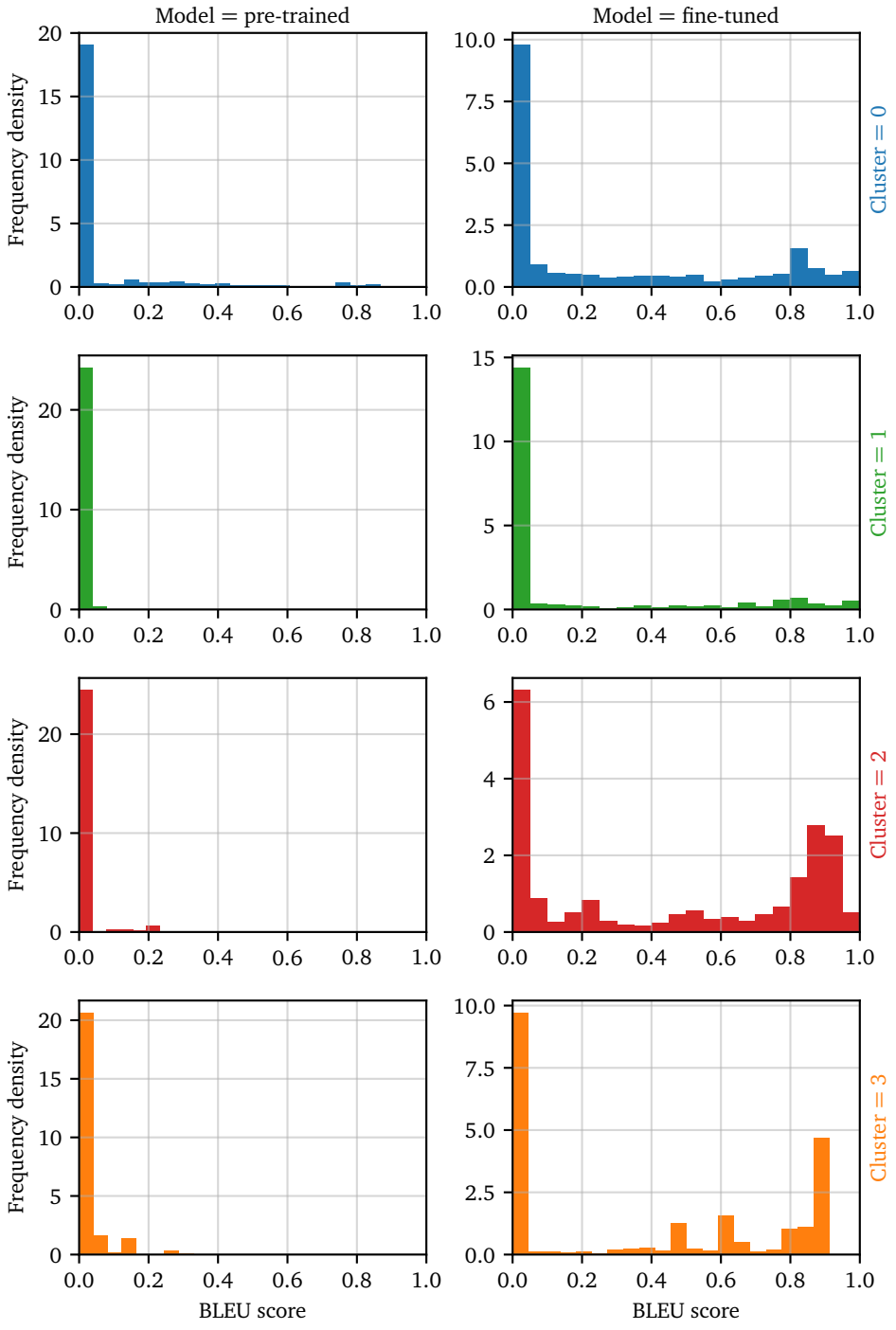


Figure 6.1: BLEU score frequency distribution of generated functions grouped by model and comment cluster, using only comments as model input.

the fine-tuned model (Figure 6.3) are much better. The number of samples with a BLEU score close to 0 is more than half compared to the pre-trained model. The rest of the scores resemble a normal distribution skewed towards the far right, peaking around a score of 0.85. This is a very good sign that the fine-tuned model performs well. Averaging the BLEU scores for each of the two models, the pre-trained model achieves a score of 0.258, while the fine-tuned model achieves 0.557. This is over a 100% improvement from the pre-trained model.

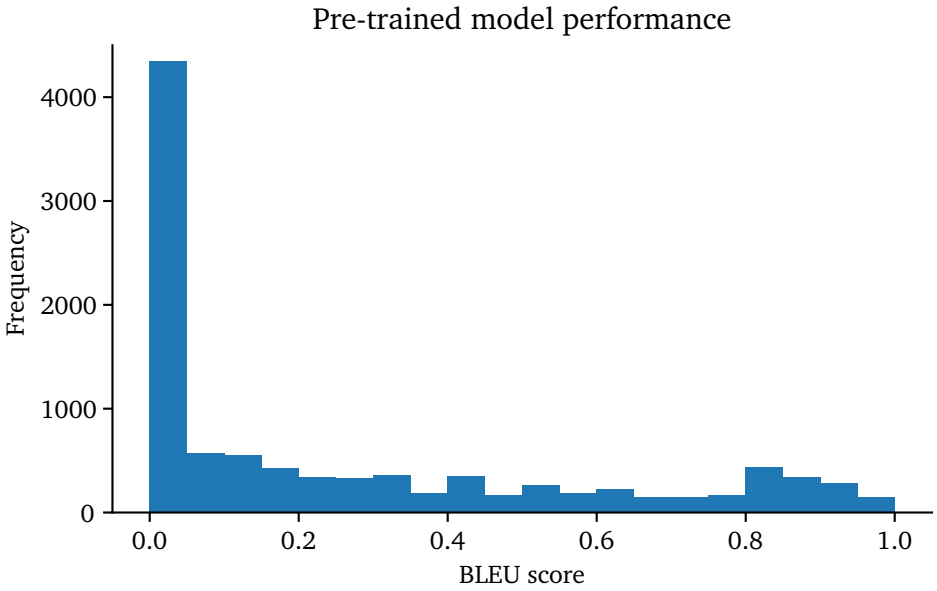


Figure 6.2: BLEU score frequency distribution of 10.000 generated functions with pre-trained model using comment-aided approach.

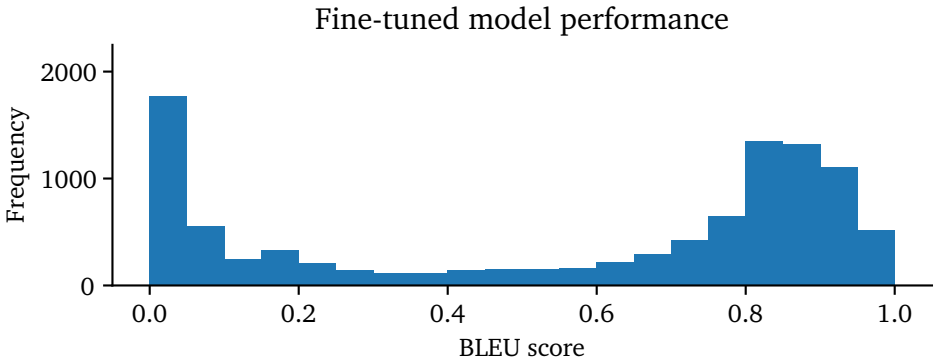


Figure 6.3: BLEU score frequency distribution of 10.000 generated functions with fine-tuned model using comment-aided approach.

6.2 Evaluation of RQ2

For evaluating the implementation for research question 2, it is analyzed how secure the generated code is. This is done by comparing the security of a fine-tuned model with and without utilizing security conditioning purposed in Section 4.5.1. However, first an evaluation of potential performance degradation is done in Section 6.2.1. Then the method used for the security evaluation is described in detail in Section 6.2.2. Finally, the results are presented in Sections 6.2.3 and 6.2.4.

6.2.1 Performance degradation evaluation

For ensuring the security conditioning method does not affect the performance of the code generation, the same evaluation procedure as for the fine-tuned model in Section 6.1.4 is done. The "<|secure|>" label is prepended to the input, and fed to the model for code generation. The result can be seen in Figure 6.4. Compared to Figure 6.3, there is no significant difference in the distribution of the BLEU score. Further, the average BLEU score is also very close to that of the pre-trained model, measuring 0.554 instead of 0.557. This is a negligible difference, meaning the security conditioning method does not degrade the model performance.

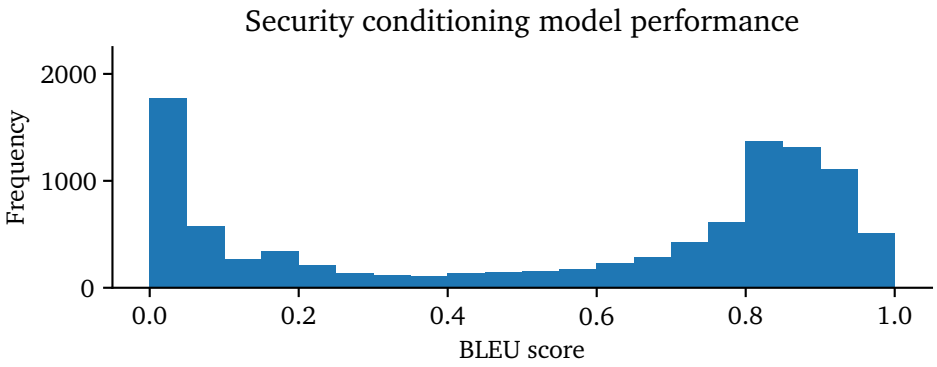


Figure 6.4: BLEU score frequency distribution of 10.000 generated functions with fine-tuned model with security conditioning using comment-aided approach.

6.2.2 Security evaluation method

For evaluating how secure the generated outputs are, this project use counting as the evaluation metric. The number of vulnerabilities introduced by the generated code is simply counted and compared to the number of vulnerabilities in the original code. The conceptual method builds upon the one used for evaluating RQ1 (see Section 6.1.1), and is as follows: Comment + code context is sampled from a contract. The code for the next function is then generated with both the fine-tuned model developed for RQ1 and the fine-tuned model with security conditioning developed for RQ2. Since security conditioning uses two labels "<|secure|>"

and "<|vulnerable|>" for distinguishing secure and vulnerable code, the model should technically also be capable of generating vulnerable code. Hence, for evaluation, both configurations are tested. The model w/ security conditioning using "<|secure|>" will be addressed as the "secure" model, and the model w/o security conditioning using "<|vulnerable|>" will be addressed as the "vulnerable" model. The three results are then run through SolDetector [72] for vulnerability analysis, and the results are compared.

Section 6.2.3 presents the evaluation results using the method above on real contracts from the test split of the Verified Smart Contract Code Comments dataset, using all available code context (supporting code). In addition, a manual evaluation is performed in Section 6.2.4, using much of the same method as above to further validate the results.

6.2.3 Comment + code context evaluation

For evaluating the security conditioning method, a total of 10.000 random samples are drawn from the test split of the Verified Smart Contract Code Comments dataset. The samples are then evaluated according to the method described in Section 6.2.2. Figure 6.5 shows the number of vulnerabilities from the evaluation, grouped by vulnerability severity. As can be seen from the figure, the number of vulnerabilities is very high. Comparing the fine-tuned model with the secure model, one can see a tiny reduction of vulnerabilities using the model w/ security conditioning. Comparing the fine-tuned model with the vulnerable model, a tiny increase in the number of vulnerabilities can be seen. Figure 6.6 shows the difference in the number of vulnerabilities produced by the secure and vulnerable models, relative to those produced by the fine-tuned model. The secure model produces 13 fewer high-risk vulnerabilities than the fine-tuned model, while the vulnerable model produces 23 more. Compared to the actual count of vulnerabilities, the difference is not significant. However, as discovered in Section 5.2.1.2, almost 50% of the contracts used for evaluation contain high-risk vulnerabilities. Depending on the amount of code context used as input for code generation, it may be hard for the model to avoid introducing vulnerabilities, as the code context is heavily biased. For example, the generated function might (have to) make use of a function that is vulnerable.

Since the security conditioning model only labels high-risk vulnerabilities as vulnerable, it is not expected to see much difference in the number of vulnerabilities for medium- and low-risk vulnerabilities. However, as can be seen from Figure 6.6, there are some differences here. Especially for medium-risk vulnerabilities, the vulnerable model produces 87 more vulnerabilities compared to the fine-tuned model. This might be due to the fact that many of the contracts with high-risk vulnerabilities also contain a lot of medium- and low-risk vulnerabilities, as can be seen in Figure 5.12.

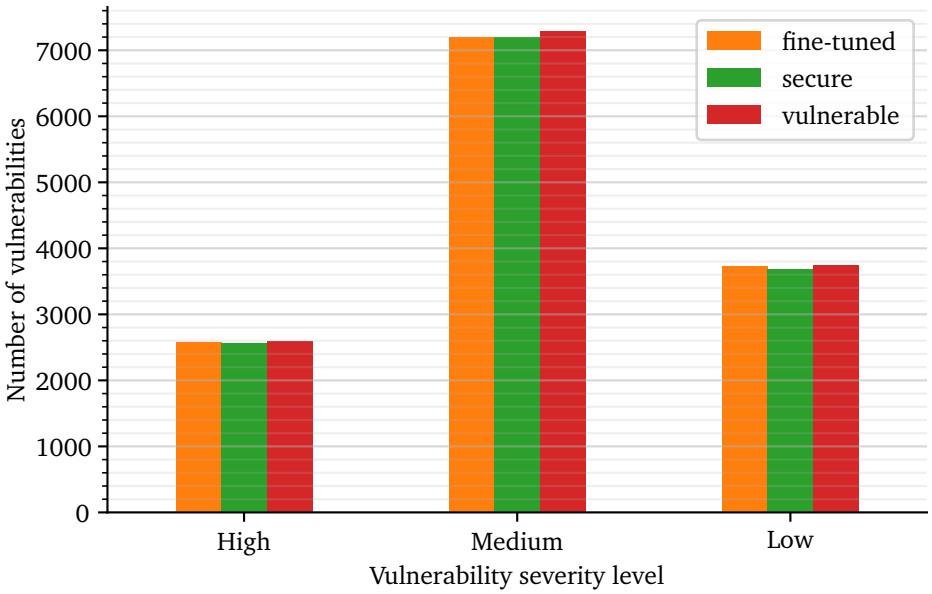


Figure 6.5: Count of vulnerabilities.

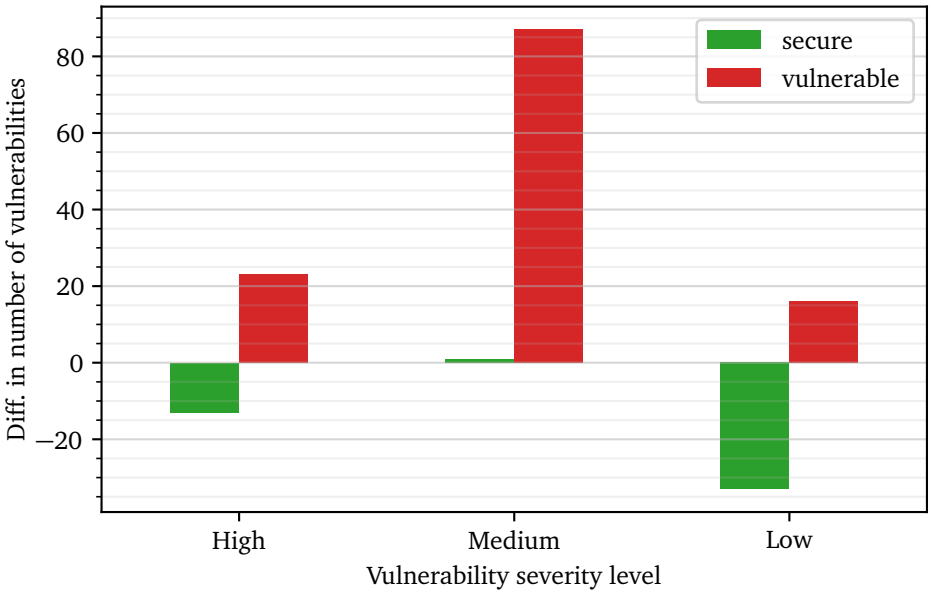


Figure 6.6: Difference in count of vulnerabilities compared to fine-tuned model without security conditioning.

6.2.4 Manual evaluation

A manual analysis of the performance of the security conditioning is performed. Several different prompts are tested. This includes code snippets and comments, both in combination and in isolation. All prompts are rather short to account for the narrow search space of long contracts, as discussed in Section 6.2.3. Further, the prompts are designed to potentially produce vulnerabilities. This way, one can get some insights into whether or not the trend seen in Figure 6.6 is in fact due to the use of security conditioning. The prompts are fed to both the fine-tuned model and the secure model.

Following are some of the most interesting code examples presented as listings. The listing is separated into two parts. The first part is the provided prompt to the two models, and the second part shows the generated code in the form of a code diff. The code from the fine-tuned model serves as the original code in the diff, and code from the secure model is shown as the code change. Removals are shown with red background, additions are shown with green background.

Experiments related to integer overflow vulnerability. Code listing 6.2 shows the diff between the fine-tuned model and the secure model generated output. As can be seen, the fine-tuned model blindly does an arithmetic addition operation on the "potential" state variable `amount` on line 5, resulting in a potential integer overflow vulnerability. The secure model avoids this by using an `add` function in line 6. This function is most likely taken from the popular SafeMath library, a library that provides that provide functions for doing safe arithmetic operations. 25 different prompts were tested. About half of the generated code contained a potential integer overflow or underflow when using the fine-tuned model. In contrast, the secure model had almost eliminated this vulnerability.

Code listing 6.2: Integer overflow vulnerability evaluation example.

```
1    /**
2     * @dev Increment amount by 'amt'.
3     */
4    function incAmount(uint256 amt) public {
5  -     amount += amt;
6  +     amount = amount.add(amt);
7    }
```

Experiments related to reentrancy vulnerability. Code listing 6.3 shows the diff between the fine-tuned model and the secure model generated output that looks like a potential reentrancy vulnerability. While the generated code is mostly identical, the order of one line of code concerned with updating the balance of the user is different. As can be seen from the listing, the fine-tuned model adds this line *after* the transfer of Ether in line 5, whereas the secure model does this *before*

in line 3. Updating the balance of the user is a critical operation, as it is used to determine the amount of Ether that can be transferred. If this is done after a transfer has been made, the caller could be able to issue a reentrancy attack. However, doing the balance update before the transfer resolves this problem. This ordering of state updates is observed multiple times. However, most of the time both models use the `transfer` function instead of the `call` function in line 4. The `transfer` function imposes a gas limit of 2300 units, which is usually enough to block the caller from issuing a reentrancy attack.

Code listing 6.3: Reentrancy vulnerability evaluation example.

```
1     function withdrawFunds (uint256 _weiToWithdraw) public {
2         require(balances[msg.sender] >= _weiToWithdraw);
3   +     balances[msg.sender] = balances[msg.sender].sub(_weiToWithdraw);
4         msg.sender.call.value(_weiToWithdraw);
5   -     balances[msg.sender] = balances[msg.sender].sub(_weiToWithdraw);
6
7     }
```

Experiments related to unchecked send vulnerability. Send calls do not revert if an exception occurs during the execution of the transfer of Ether. Instead, they return a boolean value indicating whether it is completed successfully or not. Hence, the return value of these calls should always be checked before updating state values. For checking how the system handles unchecked send calls, the generation is forced to include the word "send" in the generated code. This requires using a beam search for the generation and is set to use 5 beams. The upper half of Code listing 6.4 shows a rather large part of a "Lottery" contract is used as input to the models. The input contains the start of a function, forcing it to complete it. The lower part of the listing shows the diff. As can be seen on line 17, the fine-tuned model uses the `send` function for withdrawing the contract balance. As the secure model circumvents the using the `send` function requirement by combining "send" with "er", resulting in being able to use the much safer `transfer` function on line 19, avoiding the unchecked send vulnerability all together. The same characteristics are seen for several other input variations.

Code listing 6.4: Unchecked send vulnerability evaluation example.

```
1     contract Lottery {
2         bool public payedOut = false;
3         address public winner;
4         uint public winAmount;
5
6         /// @dev Transfer winAmount to winner
7         function payWinner() public {
8             require(!payedOut);
```

```
9             winner.transfer(winAmount);
10            payedOut = true;
11        }s
12
13        /// @dev Empty leftover funds.
14        function withdrawLeftOver() public {
15            require(payedOut);
16  -         require(winner!= address(0));
17  -         winner.send(address(this).balance);
18  +         require(msg.sender == winner);
19  +         msg.sender.transfer(address(this).balance);
20            winAmount = 0;
21        }
22    }
```

Chapter 7

Discussion

In this chapter, the results of the implementation and evaluation given in Chapters 5 and 6 are discussed. First, the transformer model fine-tuned fine-tuned for SC code generation is discussed in Section 7.1. Then, the security conditioning approach developed for answering research question 2 is discussed in Section 7.1.4.

7.1 Discussion of RQ1 Results

According to research question 1, this thesis has investigated how to automatically generate Smart Contract code with transformer-based language models, by inputting comments to guide the code generation. In the following sections, the results of the implementation and evaluation of research question 1 are discussed.

7.1.1 Comparison with related work

For answering the first part of the research question, one of the largest open-source transformer models was fine-tuned on real Ethereum SCs. The implementation achieves an accuracy of 0.917 and perplexity of 1.510. This is a significant improvement compared to the pre-trained model, which achieves an accuracy of 0.800 and a perplexity of 2.600. The rather high accuracy from pre-training is most likely due to the high percentage of comments in the dataset, many of which are written in natural language.

A side product of the fine-tuned model is the construction of the currently largest dataset of real SC ever created, consisting of 186.397. The largest competing dataset [78] contains 45,622 real-world SCs, filtered down from 1.5 million contracts by comparing the MD5 hash of the contracts. However, since they fail to inflate the contracts to remove library code, it is unfit for use in deep learning applications.

The pre-trained model and the fine-tuned model were then evaluated in Chapter 6. In Section 6.1.4, the comment-aided approach was evaluated. Firstly, the fine-tuning process of the GPT-J-6B from EleutherAI results in a BLEU score of

0.557, an increase of over 100%, up from 0.258. This is a significant improvement and can provide developers with substantial help for constructing SCs. In Section 6.1.3, the standard approach used by e.g. [2, 4] was evaluated, using only comments for generating code. As discovered, the best performing clusters (2 and 3) contained a lot of library code. As a lot of smart contract code is the implementation of libraries, the fine-tuned model could be a valuable resource for SC developers. For comparing the two approaches, the average BLEU score from clusters 0 and 1 is used. This results in an average score of 0.034 for the pre-trained model, and 0.2245 for the fine-tuned model. Comparing the results from the two approaches, the comment-aided approach produces significantly better results. The pre-trained model reports a staggering 770% increase in the BLEU score, and a fine-tuned model reports a 248% increase. This clearly shows the power of the comment-aided code generation approach.

Compared to other works reporting BLEU score as metric, the results from this thesis outperform the state-of-the-art. For example, PyMT5 achieves a bleu score of 0.0859 [4]. Unfortunately, more recent works like Codex and AlphaCode do not evaluate their model using BLEU score but use functional correctness instead (discussed in Section 6.1.2). Codex evaluates the functional correctness performance of the pre-trained GPT-J-6B model. They report GPT-J solves 11.6% Python problems [2], while Codex solves 28.8% [2]. As this work achieves over 100% improvement in the BLEU score from the pre-trained model, it is not unreasonable to expect at least similar results to Codex, a 12 Billion parameter model (twice the size of GPT-J).

7.1.2 Implication to academia and industry

Several of the results from research question 1 can have a major impact on both academia and industry. First, the transformer model fine-tuned for SC code generation can rather accurately generate SC code. Using this model in an industry setting has the potential to greatly reduce the efforts needed for creating SCs. It can also help reduce the expertise level required for the development of SCs, as the contract can in large parts be generated from natural language comment description. As a lot of smart contract code is the implementation of libraries, the fine-tuned model could be a valuable resource for SC developers.

As code-synthesis using transformers is a rather new area, not a lot of attention has been devoted to exploring *how* to best make use of these systems. This thesis proposes a novel comment-aided approach for guiding the code generation. As discussed in Section 7.1.1, using this approach greatly increase the performance of the model. This method can most likely also be applied to other programming languages and models, greatly increasing the performance of such solutions. Further, these results are a great motivate further research, investigating other ways to guide code generation.

7.1.3 Threats to validity

Considering internal threats to validity, the main threat is cross-contamination between the splits of the datasets. this applies to both the performance of the fine-tuned model, as well as the evaluation of the comment-aided code generation approach. As described in Section 5.1.1, a lot of the SCs contains duplicated code. If these duplications make up a too large percentage of the dataset, this would lead to overly-optimistic estimates of the model's performance. To reduce the likelihood of this, several counter measurements are taken. Primarily, extensive filtering of the contracts based on similarity is done in Section 5.1.1.1. To further aid this filtering, the contract files were inflated, as described in Section 5.1.1.2. As over 5 million contracts are filtered down to 186.397 contracts, it can be assumed that the risk of cross-contamination is rather low.

Concerning the external validity of the comment-aided code generation approach, it is likely that the approach can be used for other programming languages as well as other models. The approach does not rely on any specifics of SC language. However, the approach requires the code context used to be relevant to the comment. As SCs are smaller and less complex than other languages, there might be some restrictions on the generalizability of the approach. For example, trying to use the approach to generate a Python function from a comment using a completely unrelated function as code context will probably not work well.

7.1.4 Discussion of RQ2 results

According to research question 2, this thesis has investigated how to automatically generate secure Smart Contract code with transformer-based language models. For answering this, a novel security conditioning technique was developed. In the following sections, the results of the implementation and evaluation of research question 2 are discussed.

7.1.5 Comparison with related work

For testing the technique, the pre-trained model used in RQ1 was fine-tuned using security conditioning. The implementation achieves similar scores to that achieved in RQ1, with an accuracy of 0.917 and perplexity of 1.510. Further, the code generation performance of the fine-tuned model using the security conditioning technique is compared to that of the fine-tuned model in Section 6.2.1. As reported, the variations in BLEU score are negligible, with a difference of 0.003. Hence, the technique introduces no performance degradation.

The security of the code generated using the security conditioning technique is evaluated in Section 6.2.3. First, the technique is automatically evaluated using the comment-aided approach. The results of the evaluation indicate that the security conditioning technique does work. However, the reduction of vulnerabilities compared to the total number of vulnerabilities is rather low. As described in Section 6.2.3, this might be because the code used as context already contains a lot

of vulnerabilities. Therefore, some manual tests were performed in Section 6.2.4. The manual testing further strengthens the findings from the automatic evaluation. For example, the model with security conditioning produces significantly fewer "integer overflow and underflow" vulnerabilities. As shown in Figure 5.13, this is also by far the most common SC vulnerability. For the less common vulnerabilities such as "reentrancy" and "unchecked send" vulnerabilities, the differences are more subtle. It is rather hard to make any of the models generate one of these less common high-risk vulnerabilities. However, when sufficiently provoked to generate a vulnerability, the model with security conditioning almost always generates a safe alternative.

As presented in Section 3.2, works in other domains have tried to reduce bias in language models. However, with mixed results. In contrast, the security conditioning technique does seem to work, without introducing any performance degradation.

7.1.6 Implication to academia and industry

As reported by Pearce *et al.* [35], existing code synthesis solutions based on transformers produce a lot of vulnerabilities. Being able to reduce this number would be of great value to the industry. This applies especially to the creation of SCs, as vulnerabilities can not be fixed after it has been deployed to the blockchain. From the evaluation in Section 6.2, the model with security conditioning can be used to generate mostly SCs code.

The security conditioning technique can also serve as the basis for a lot of further research. Not only for vulnerability analysis but also in other areas. As discussed in Section 3.2 vulnerabilities can be considered as a form of bias in language models. Hence, the technique could maybe be generalized to handle other types of biases, for example, gender bias. Since the technique primarily relies on augmenting the dataset, the method should also be transferable to other models as well.

From the results of the vulnerability analysis of the SC datasets in Section 5.2.1.2, a very large percentage of the SCs contracts are vulnerable. This is a very interesting finding and can both aid and motivate future research on SC vulnerabilities.

7.1.7 Threats to validity

As seen from the evaluation of Sections 6.2.3 and 6.2.4, evaluating the security of the model is very hard. In most of the manual generation examples, except for "integer overflow and underflow" vulnerabilities, neither of the two models produces any vulnerabilities. This could either be that the prompts are not sufficiently complex, or that the model is already secure. Pearce *et al.* [35] is one of very few works that investigate the security of transformer-based code generation. They conduct a vulnerability analysis of GitHub Copilot (based on Codex), reporting approximately 40% of 1689 synthesized Python and C programs to be vulnerable. It is therefore not very likely that the fine-tuned developed for RQ1 mostly

produces secure code. To evaluate the security conditioning technique more thoroughly, one could adopt the same approach as used by [35]. This would require the creation of a manual dataset, where the code is carefully crafted so that it *may* produce a vulnerability.

Another potential threat is the quality of the labeled data. The security conditioning technique is only as good as the labeled data. For labeling the SCs, SolDetector was the only vulnerability detection tool used. If SolDetector incorrectly labels a SCs as vulnerable, this could confuse the model.

Chapter 8

Conclusion and Future Work

In this thesis, ways for generating Smart Contract (SC) code with transformer models have been explored. This includes both how to guide the code generation by inputting comments, as well as how to generate secure SC code. In this chapter, the results and findings from the thesis are concluded in Section 8.1, along with potential future works in Section 8.2.

8.1 Conclusion

To generate Smart Contract (SC) code with a transformer model, this thesis fine-tunes a state-of-the-art open-source 6 billion parameter transformer model on SC code. To facilitate the training of this large model, the currently largest dataset of SCs is constructed, containing over 186,397 real verified SCs. Further, this thesis proposes a novel comment-aided approach to guide the code synthesis. From evaluating the approach by generating 10.000 functions with the fine-tuned model, a BLEU score of 0.557 is achieved, outperforming the state-of-the-art.

In order to produce secure SC code with a transformer model, this thesis proposes a novel security technique named security conditioning. By using special tokens as labels during the training of a transformer model, it can condition on secure or vulnerable code. As this technique requires the SCs to be labeled as secure or vulnerable, the 186,397 SC are labeled with a vulnerability detection tool named SolDetector, resulting in the currently largest audited SC dataset. In this thesis, the effectiveness of security conditioning is demonstrated by fine-tuning a transformer model using security conditioning. From both automatic and manual evaluation of the technique, there are good indications that security conditioning does produce fewer vulnerabilities without performance degradation. Especially improvement is seen for common vulnerabilities, such as integer overflows and underflows.

To summarize, this thesis is the first to generate SC code with transformer-based language models. Further, the code generation is improved by using a novel comment-aided approach, achieving state-of-the-art results. Finally, the security of the generated code is improved by a novel security conditioning technique.

8.2 Future work

There are several interesting ideas and observations arising from this thesis. Following are some potential improvements and suggestions of topics for future work:

- Conduct a user study on the effects of using the comment-aided approach for code generation.
- Evaluate how hyperparameter tuning affects security conditioning.
- Use multiple vulnerability detection tools for labeling SC and study how it affects performance.
- How to automatically evaluate *potential* security vulnerabilities in synthesized code.

Bibliography

[1] GitHub. "Your ai pair programmer." (2022), [Online]. Available: https://github.com/features/copilot (visited on 07/07/2022).

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, *Evaluating large language models trained on code*, 2021. DOI: 10.48550/ARXIV.2107.03374. [Online]. Available: https://arxiv.org/abs/2107.03374.

[3] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, *Competition-level code generation with alphacode*, 2022. DOI: 10.48550/ARXIV.2203.07814. [Online]. Available: https://arxiv.org/abs/2203.07814.

[4] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: Multi-mode translation of natural language and python code with transformers," *CoRR*, vol. abs/2010.03150, 2020. arXiv: 2010.03150. [Online]. Available: https://arxiv.org/abs/2010.03150.

[5] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*, Springer, 2017, pp. 164–186.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, 2017. DOI: 10.48550/ARXIV.1706.03762. [Online]. Available: https://arxiv.org/abs/1706.03762.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2018. DOI: 10.48550/ARXIV.1810.04805. [Online]. Available: https://arxiv.org/abs/1810.04805.

[8] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[9] HuggingFace. "Accuracy." (Apr. 2022), [Online]. Available: https://huggingface.co/spaces/evaluate-metric/accuracy (visited on 06/22/2022).

[10] HuggingFace. "Perplexity of fixed-length models." (Apr. 2022), [Online]. Available: https://huggingface.co/docs/transformers/perplexity (visited on 06/22/2022).

[11] S. Goyalm. "Perplexity of fixed-length models." (Sep. 2019), [Online]. Available: https://medium.com/analytics-vidhya/no-need-to-be-perplexed-by-perplexity-cd4cb71ac97b (visited on 06/22/2022).

[12] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02, Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. [Online]. Available: https://doi.org/10.3115/1073083.1073135.

[13] Google. "Understanding the bleu score." (Jul. 2022), [Online]. Available: https://cloud.google.com/translate/automl/docs/evaluate#bleu (visited on 07/14/2022).

[14] A. Lavie. "Evaluating the output of machine translation systems." (Sep. 2011), [Online]. Available: https://www.cs.cmu.edu/~alavie/Presentations/MT-Evaluation-MT-Summit-Tutorial-19Sep11.pdf (visited on 07/14/2022).

[15] DeepAi. "What is the jaccard index?" (Apr. 2022), [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/jaccard-index (visited on 06/28/2022).

[16] Ethereum. "Gas and fees." (Dec. 2021), [Online]. Available: https://ethereum.org/en/developers/docs/gas/ (visited on 01/04/2022).

[17] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *International Conference on Machine Learning*, Aug. 2015, pp. 2123–3132. [Online]. Available: https://www.microsoft.com/en-us/research/publication/bimodal-modelling-of-source-code-and-natural-language/.

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, Zurich, Switzerland: IEEE Press, 2012, pp. 837–847, ISBN: 9781467310673.

[19] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," in *Proceedings of ICLR'17*, Mar. 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/.

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, *Code2vec: Learning distributed representations of code*, 2018. DOI: 10.48550/ARXIV.1803.09473. [Online]. Available: https://arxiv.org/abs/1803.09473.

[21] U. Alon, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code," *CoRR*, vol. abs/1808.01400, 2018. arXiv: 1808.01400. [Online]. Available: http://arxiv.org/abs/1808.01400.

[22] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: AI-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, Jul. 2019. DOI: 10.1145/3292500.3330699. [Online]. Available: https://doi.org/10.1145%2F3292500.3330699.

[23] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, *Deep contextualized word representations*, 2018. DOI: 10.48550/ARXIV.1802.05365. [Online]. Available: https://arxiv.org/abs/1802.05365.

[24] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, *Xlnet: Generalized autoregressive pretraining for language understanding*, 2019. DOI: 10.48550/ARXIV.1906.08237. [Online]. Available: https://arxiv.org/abs/1906.08237.

[25] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, *Roberta: A robustly optimized bert pretraining approach*, 2019. DOI: 10.48550/ARXIV.1907.11692. [Online]. Available: https://arxiv.org/abs/1907.11692.

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, *Codebert: A pre-trained model for programming and natural languages*, 2020. DOI: 10.48550/ARXIV.2002.08155. [Online]. Available: https://arxiv.org/abs/2002.08155.

[27] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, *Codesearchnet challenge: Evaluating the state of semantic code search*, 2019. DOI: 10.48550/ARXIV.1909.09436. [Online]. Available: https://arxiv.org/abs/1909.09436.

[28] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, *Intellicode compose: Code generation using transformer*, 2020. DOI: 10.48550/ARXIV.2005.08025. [Online]. Available: https://arxiv.org/abs/2005.08025.

[29] Codeforces. "Codeforces." (), [Online]. Available: https://codeforces.com/ (visited on 05/01/2022).

[30] B. Li, H. Peng, R. Sainju, J. Yang, L. Yang, Y. Liang, W. Jiang, B. Wang, H. Liu, and C. Ding, *Detecting gender bias in transformer-based models: A case study on bert*, 2021. DOI: 10.48550/ARXIV.2110.15733. [Online]. Available: https://arxiv.org/abs/2110.15733.

[31] A. Silva, P. Tambwekar, and M. C. Gombolay, "Towards a comprehensive understanding and accurate evaluation of societal biases in pre-trained transformers," in *NAACL*, 2021.

[32] D. Madras, E. Creager, T. Pitassi, and R. Zemel, "Learning adversarially fair and transferable representations," Feb. 2018.

[33] B. H. Zhang, B. Lemoine, and M. Mitchell, *Mitigating unwanted biases with adversarial learning*, 2018. DOI: 10.48550/ARXIV.1801.07593. [Online]. Available: https://arxiv.org/abs/1801.07593.

[34] S. Hofstätter, A. Lipani, S. Althammer, M. Zlabinger, and A. Hanbury, *Mitigating the position bias of transformer models in passage re-ranking*, 2021. DOI: 10.48550/ARXIV.2101.06980. [Online]. Available: https://arxiv.org/abs/2101.06980.

[35] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, *Asleep at the keyboard? assessing the security of github copilot's code contributions*, 2021. DOI: 10.48550/ARXIV.2108.09293. [Online]. Available: https://arxiv.org/abs/2108.09293.

[36] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017, ISSN: 2325-1107. DOI: 10.1561/2500000010. [Online]. Available: http://dx.doi.org/10.1561/2500000010.

[37] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language models are few-shot learners*, 2020. DOI: 10.48550/ARXIV.2005.14165. [Online]. Available: https://arxiv.org/abs/2005.14165.

[38] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, *Asleep at the keyboard? assessing the security of github copilot's code contributions*, 2021. DOI: 10.48550/ARXIV.2108.09293. [Online]. Available: https://arxiv.org/abs/2108.09293.

[39] Z. Smith, E. Lostri, and M. (Firm), *The hidden costs of cybercrime*, 2020. [Online]. Available: https://www.csis.org/analysis/hidden-costs-cybercrime.

[40] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186, ISBN: 978-3-662-54455-6.

[41] B. J. Oates, *Researching Information Systems and Computing*. Sage Publications Ltd., 2006, ISBN: 1412902231.

[42] V. K. Vaishnavi and W. L. Kuechler, "Design Science Research in Information Systems," *Ais*, pp. 1–45, 2004, ISSN: 02767783. DOI: 10.1007/978-1-4419-5653-8. [Online]. Available: http://www.desrist.org/design-research-in-information-systems/.

[43] Ethereum. "Natspec format." (Feb. 2022), [Online]. Available: https://docs.soliditylang.org/en/v0.8.15/natspec-format.html (visited on 05/10/2022).

[44] B. Wang and A. Komatsuzaki, *GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model*, https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[45] M. Woolf. "Fun and dystopia with ai-based code generation using gpt-j-6b." (Jun. 2021), [Online]. Available: https://minimaxir.com/2021/06/gpt-j-6b/.

[46] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, *The pile: An 800gb dataset of diverse text for language modeling*, 2021. DOI: 10.48550/ARXIV.2101.00027. [Online]. Available: https://arxiv.org/abs/2101.00027.

[47] EleutherAI, *Download all github repositories*, 2020. [Online]. Available: https://github.com/EleutherAI/github-downloader (visited on 03/10/2022).

[48] EleutherAI. "Eleutherai." (Apr. 2022), [Online]. Available: https://www.eleuther.ai (visited on 06/28/2022).

[49] J. Su, Y. Lu, S. Pan, B. Wen, and Y. Liu, *Roformer: Enhanced transformer with rotary position embedding*, 2021. DOI: 10.48550/ARXIV.2104.09864. [Online]. Available: https://arxiv.org/abs/2104.09864.

[50] Microsoft, *Deepspeed*, version 0.6.4, 2022. [Online]. Available: https://www.deepspeed.ai/ (visited on 05/06/2022).

[51] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 530–541, ISBN: 9781450371216. DOI: 10.1145/3377811.3380364. [Online]. Available: https://doi.org/10.1145/3377811.3380364.

[52] H. Face, *Transformers*, version 4.19.0.dev0, 2022. [Online]. Available: https://www.antlr.org/index.html (visited on 06/10/2022).

[53] H. Face. "Hugging face - the ai community building the future." (Jun. 2022), [Online]. Available: https://huggingface.co/ (visited on 07/10/2022).

[54] Microsoft. "Microsoft." (Jun. 2022), [Online]. Available: https://www.microsoft.com/about (visited on 07/10/2022).

[55] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, *Zero: Memory optimizations toward training trillion parameter models*, 2019. DOI: 10.48550/ARXIV.1910.02054. [Online]. Available: https://arxiv.org/abs/1910.02054.

[56] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, *Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning*, 2021. DOI: 10.48550/ARXIV.2104.07857. [Online]. Available: https://arxiv.org/abs/2104.07857.

[57] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, *Mixed precision training*, 2017. DOI: 10.48550/ARXIV.1710.03740. [Online]. Available: https://arxiv.org/abs/1710.03740.

[58] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, *Epic: An energy-efficient, high-performance gpgpu computing research infrastructure*, 2019. DOI: 10.48550/ARXIV.1912.05848. [Online]. Available: https://arxiv.org/abs/1912.05848.

[59] N. H. P. C. Group. "Idun." (2022), [Online]. Available: https://www.hpc.ntnu.no/idun/ (visited on 07/01/2022).

[60] Ethereum. "Vyper." (Sep. 2022), [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/languages/#vyper (visited on 06/14/2022).

[61] SafeMath, *Safemath*, 2022. [Online]. Available: https://docs.openzeppelin.com/contracts/3.x/utilities#math (visited on 06/01/2022).

[62] OpenZeppelin, *Openzeppelin*, 2022. [Online]. Available: https://www.openzeppelin.com (visited on 06/01/2022).

[63] Ethereum, *Solidity grammar*, 2022. [Online]. Available: https://github.com/ethereum/solidity/tree/develop/docs/grammar (visited on 04/01/2022).

[64] T. Parr, *Antlr 4*, version 4.10.1, 2022. [Online]. Available: https://www.antlr.org/index.html (visited on 04/15/2022).

[65] T. Parr and K. Fisher, "Ll(*): The foundation of the antlr parser generator," *SIGPLAN Not.*, vol. 46, no. 6, pp. 425–436, Jun. 2011, ISSN: 0362-1340. DOI: 10.1145/1993316.1993548. [Online]. Available: https://doi.org/10.1145/1993316.1993548.

[66] ANTLR. "Runtime libraries and code generation targets." (), [Online]. Available: https://github.com/antlr/antlr4/blob/master/doc/targets.md (visited on 07/07/2022).

[67] F. Bond, *Solidity-antlr4*, 2019. [Online]. Available: https://github.com/solidityj/solidity-antlr4 (visited on 05/10/2022).

[68] ANTLR. "Visitor pattern." (), [Online]. Available: https://en.wikipedia.org/wiki/Visitor_pattern (visited on 07/07/2022).

[69] NLTK, *Nltk*, version 3.7, 2022. [Online]. Available: https://www.nltk.org (visited on 05/27/2022).

[70] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space*, 2013. DOI: 10.48550/ARXIV.1301.3781. [Online]. Available: https://arxiv.org/abs/1301.3781.

[71] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," ser. Proceedings of LREC 2010 workshop New Challenges for NLP Frameworks, Valetta, MT: University of Malta, May 2010, pp. 45–50. [Online]. Available: http://is.muni.cz/publication/884893/en.

[72] B. L. Tianyuan Hu Zhenyu Pan, "Soldetector: Detect defects based on knowledge graph of solidity smart contract," in *Proceedings of the 32rd International Conference on Software Engineering and Knowledge Engineering SEKE*, 2021, pp. 423–428.

[73] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," English, in *CCS'16: PROCEEDINGS OF THE 2016 ACM SIGSAC CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY*, 23rd ACM Conference on Computer and Communications Security (CCS), Vienna, AUSTRIA, OCT 24-28, 2016, Assoc Comp Machinery; ACM Special Interest Grp Secur Audit & Control, 1515 BROADWAY, NEW YORK, NY 10036-9998 USA: ASSOC COMPUTING MACHINERY, 2016, 254–269, ISBN: 978-1-4503-4139-4. DOI: {10.1145/2976749.2978309}.

[74] Pexpect, *Pexpect*, version 4.8.0, 2022. [Online]. Available: https://github.com/pexpect/pexpect (visited on 07/10/2022).

[75] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, *Codebleu: A method for automatic evaluation of code synthesis*, 2020. DOI: 10.48550/ARXIV.2009.10297. [Online]. Available: https://arxiv.org/abs/2009.10297.

[76] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, *Unsupervised translation of programming languages*, 2020. DOI: 10.48550/ARXIV.2006.03511. [Online]. Available: https://arxiv.org/abs/2006.03511.

[77] M. Neto. "How to issue your own token on ethereum in less than 20 minutes." (Dec. 2017), [Online]. Available: https://medium.com/bitfwd/how-to-issue-your-own-token-on-ethereum-in-less-than-20-minutes-ac1f8f022793 (visited on 07/13/2022).

[78] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: What is the best choice?" In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 566–579, ISBN: 9781450384599. DOI: 10.1145/3460319.3464837. [Online]. Available: https://doi.org/10.1145/3460319.3464837.

André Storhaug

Secure Smart Contract Code Synthesis

NTNU
Norwegian University of
Science and Technology