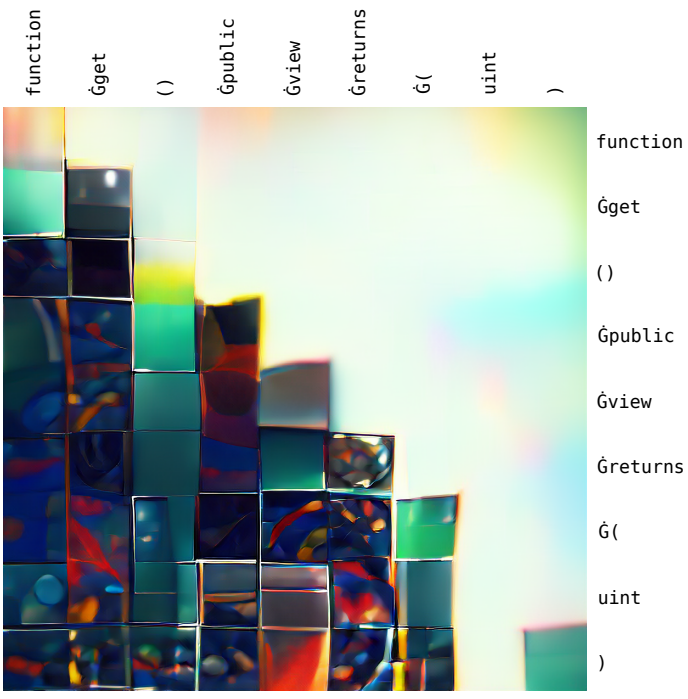
Master's thesis

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

Aute culpa cillum elit non sunt mollit tempor dolore tempor excepteur. Pariatur nostrud consequat pariatur in officia commodo tempor consequat veniam in velit. Cupidatat adipisicing eiusmod sunt laboris ex deserunt ullamco laboris. Incididunt cillum dolor aute irure id cupidatat irure. Cillum nulla mollit incididunt commodo consectetur. Est cupidatat et excepteur non ad. Esse et Lorem dolor laboris sit velit incididunt dolor veniam pariatur est ad.

Sammendrag

Aute culpa cillum elit non sunt mollit tempor dolore tempor excepteur. Pariatur nostrud consequat pariatur in officia commodo tempor consequat veniam in velit. Cupidatat adipisicing eiusmod sunt laboris ex deserunt ullamco laboris. Incididunt cillum dolor aute irure id cupidatat irure. Cillum nulla mollit incididunt commodo consectetur. Est cupidatat et excepteur non ad. Esse et Lorem dolor laboris sit velit incididunt dolor veniam pariatur est ad.

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

AST Abstract Syntax Tree. 16, 17, 55

BERT Bidirectional Encoder Representations from Transformers. 4

bfloat16 Brain Floating Point. 23, 26, 41

BLEU BiLingual Evaluation Understudy. xiv, xv, xvii, 9, 10, 17, 18, 55–60, *Glossary:* BiLingual Evaluation Understudy

BPE Byte-Pair Encoding. 40

CLM Casual Language Modeling. xvi, 23, 41, 42, 49

DSR Design Science Research. 20

EVM Ethereum Virtual Machine. xvii, 55, *Glossary:* Ethereum Virtual Machine

float16 Half-precision Floating-Point. 26

GPT General Pre-trained Transformer. 4

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

LSTM Long Short-Term Memory. 17

ML Machine Learning. 2, 14

NFT Non Fungible Tokens. xvii, 11, *Glossary:* Non Fungible Tokens

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

OOM Out of Memory. xvii, 41, *Glossary:* Out of Memory

PCA Principal Component Analysis. 36

PCFG Probabilistic context-free grammar. 16

RNN Recurrent Neural Network. 4

RoPE Rotary Position Embedding. 22, 40

SC Smart Contract. xiii, xvi, 2, 3, 11–13, 20, 28, 29, 32, 33, 40, 41, 47–51, 54–56, 59, 63–65

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

TFIDF Term Frequency–Inverse Document Frequency. 35

TOOD TODO. xviii, 17, *Glossary:* TODO

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

ZeRO Zero Redundancy Optimizer. 25, 26, 41

Glossary

BiLingual Evaluation Understudy Metric for automatically evaluating machine-translated text. xiv, xv, 9, 10, 17, 18, 55–60

docstring Python function documentation strings. 17

Ethereum Virtual Machine The runtime environment for transaction execution in Ethereum. 55

F1 Harmonic mean of precision and recall. 17

fork A blockchain that diverges into two potential paths is called a fork. 11

Non Fungible Tokens A type of token that is unique. 11

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. 41

TODO todo. 17

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, a number of 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 permeates every aspect of our lives. However, constructing such systems is a hard and time-consuming task. A number of tools and methods have been developed to increase the productivity of programmers, as well as to making 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 language syntax' 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

Is this ok?

setting. This is a very limited approach as the potential search space is enormous. This thesis investigates a comment-aided approach to generating code, analyzing both comment style and the importance of context.

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.
- Fine-tuned transformer-based language model for Smart Contract code generation.
- Fine-tuned transformer-based language model for Smart Contract code generation.
- Novel secure code generation method.
- 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. **??** describes the methods used to implement the secure code generation. **??** describes the results of the project, and Chapter 7 discuss the findings. Identified future work is presented in Chapter 8. Chapter 9 presents final remarks and concludes the thesis.

Chapter 2

Background

This chapter introduces the necessary background information for this study. First, an introduction to natural language processing is presented in Section 2.1. Then, a thorough description of the Transformer model is provided in Section 2.2. Following is Section 2.3, explaining the different metrics used in this thesis. A brief introduction to blockchain technology is provided in Section 2.4. Then, the concept of Smart Contracts (SCs) is introduced in Section 2.5. Finally, in Section 2.5.1, the most popular SC vulnerabilities are described.

2.1 Natural language processing

Used for comment clustering

2.1.1 Text preprocessing

reducing capitalization, etc... Stemming is the process of reducing a word to its word stem that affixes to suffixes and prefixes or to the roots of words known as a lemma. For example: words such as “Likes”, ”liked”, ”likely” and ”liking” will be reduced to “like” after stemming.

2.1.2 Vector space model

2.1.2.1 Word2Vec

Or word embedding? as secttion. https://ai.stackexchange.com/questions/26739/what-is-the-difference-between-a-language-model-and-a-word-embedding

Word Embeddings does not consider context, Language Models does. For e.g Word2Vec, GloVe, or fastText, there exists one fixed vector per word.

The sentence:

Kill him, don’t wait for me.

and

Don’t kill him, wait for me.

If one averages the word embeddings, they would produce the same vector. However, in reality thheir meaning (semantic) is very diffferent.

2.1.2.2 Term frequency-inverse document frequency

2.2 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 Transformerss (BERTs) and GPTs 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.

2.2.1 Attention

Finish

The self-attention mechanism is a mechanism that allows the model to learn to focus on a specific part of the input sequence. For example, cosider the following sentence: something somethin it..

By using the self-attention mechanism, the model can learn to focus on the word *it* and ignore the other words. the context of the word *it* is the word *something* and *something*. Hence, it is essential to know what *it* refers to, in order to make a good prediction for the next word.

Figure 2.1 shows the attention scores for the word *it* in the sentence *something something it...*

2.2.2 Architecture

The standard Transformer architecture, as describe by 2017, is shown in Figure 2.2. The following subsections describe the architecture of the standard Transformer model.

2.2.2.1 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.

Figure 2.1: TODO

2.2.2.2 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.

Rewrite

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. The positional encoding vector is then added to the embedding vector.

2.2.2.3 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 2.2, 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

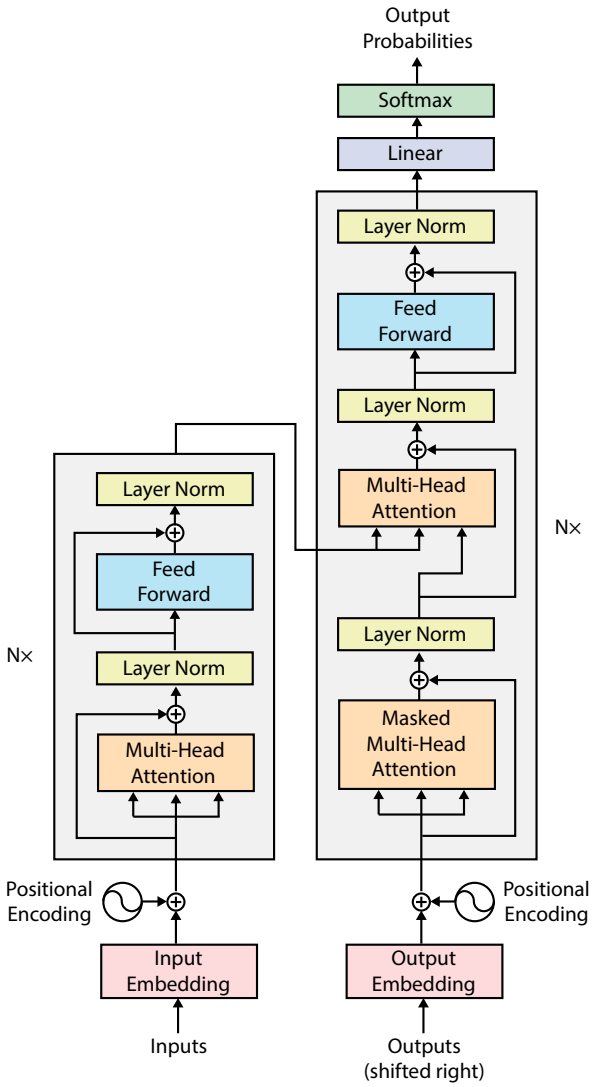


Figure 2.2: Architecture of a standard Transformer Vaswani *et al.* [6]

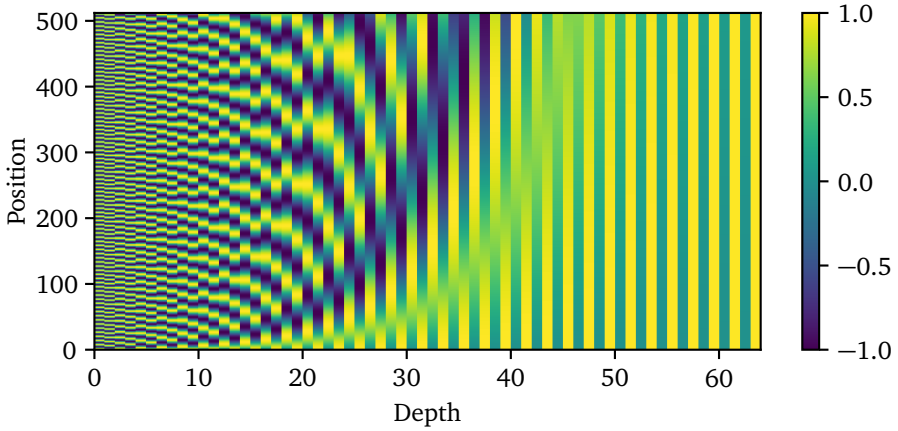


Figure 2.3: The 64-dimensional positional encoding for a sentence with the maximum length of 512. Each row represents the embedding vector p_t

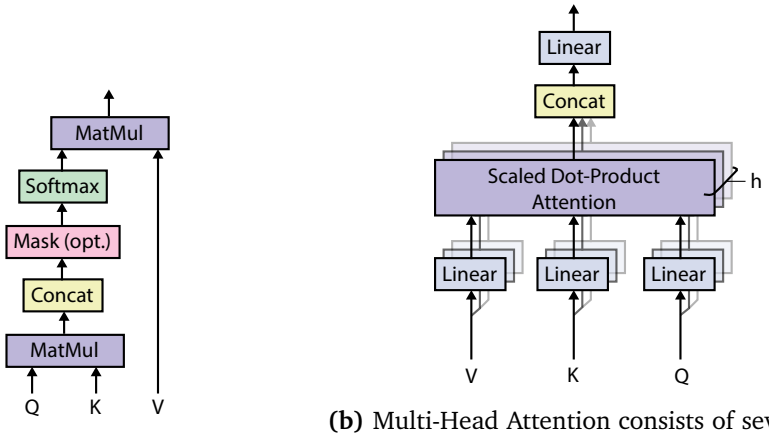
input sentence as it is handling a specific word. Each decoder layer has an additional attention mechanism that draws information from the outputs of previous decoders, before the decoder layer draws information from the encodings. Both the encoder and decoder layers contain a feed-forward layer for 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.

2.2.2.4 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 2.4a. 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



(a) Scaled Dot-Product Attention.

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

Figure 2.4: Multi-Head Attention module in Transformer architecture Vaswani *et al.* [6]

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 the softmax score. The resulting weighted 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. This significantly speeds up the training process. The queries, keys, and values are packed into separate matrices. The output matrix can be described as:

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

T means transpose. Since this is matrix calculations, do I need to add an explanation of this?

2.2.2.5 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 2.4b. 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.

2.2.3 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.

2.2.4 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.

More concrete example to explain this. Maybe a figure?

2.3 Metrics

2.3.1 Performance metric

2.3.1.1 Accuracy

2.3.1.2 Perplexity

2.3.2 Machine translation metrics

2.3.2.1 BLEU

BLEU (BiLingual Evaluation Understudy) by Papineni *et al.* [7] 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 you can achieve. 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. 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. In addition, positive weights w_n are used, normally set to $w_n = 1/N$.

$$\text{Geometric Average Precision } (N) = \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{2.2}$$

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

$$\text{Brevity Penalty} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{2.3}$$

The final BLEU score is then computed as:

$$\text{BLEU} = \text{Brevity Penalty} \cdot \text{Geometric Average Precision Scores } (N) \tag{2.4}$$

2.3.3 String metric

2.3.3.1 Jaccard index

TODO: Used for smart contract filtering

2.4 Blockchain

The blockchain technology was popularized by Bitcoin in 2008. Satoshi Nakamoto introduced the formal idea of blockchain as a peer-to-peer electronic cash system. It enabled users to conduct transactions without the need for a central authority. A blockchain is a growing list of records that are linked together by a cryptographic hash. Each record is called a block. The blocks contain a cryptographic hash of the previous block, a timestamp, and transactional data. By time-stamping a block, this proves that the transaction data existed when the block was published in order to get into its hash. Since all blocks contains the hash of the previous block, they end up forming a chain. In order to tamper with a block in the chain, this also requires altering all subsequent blocks. Blockchains are therefore resistant to modification. The longer the chain, the more secure it is.

Typically, blockchains are managed by a peer-to-peer network, resulting in a publicly distributed ledger. The network is composed of nodes that are connected

to each other. The nodes collectively adhere to a protocol in order to communicate and validate new blocks. Blockchain records are possible to alter through a fork. However, blockchains can be considered secure by design and present a distributed computing system with high Byzantine fault tolerance [8].

From Bitcoin sprang several other cryptocurrencies and blockchain platforms such as Ethereum, Litescoin, and Ripple. **??** shows an overview of the different blockchain platforms, including the different consensus protocols, programming languages, and execution environments used. It also shows the different types of blockchains, including public, private, and hybrid.

2.4.1 Ethereum blockchain

2.5 Smart Contract

Rewrite and adapt to vulnerabilities SoliDetector can detect.

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, enabling 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. Since Ethereum's introduction of SCs, the platform has kept its market share as the most popular SC blockchain platform. Ethereum is a open, decentralized platform that allows users to create, store, and transfer digital assets. Solidity is a programming language that is used to write smart contracts in 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." [9]. The gas is used to pay for the cost of running the smart contract. This protects against malicious actors spamming the network [9]. The gas is paid in Wei, which is the smallest unit of Ethereum. 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.

2.5.1 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:

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

2.5.1.1 Integer Overflow and Underflow

Integer overflow and underflows happen when an arithmetic operation reaches the maximum or minimum size of a certain data type. In particular, multiplying or adding two integers may result in a value that is unexpectedly small, and subtracting from a small integer may cause a wrap to be an unexpectedly large positive value. For example, an 8-bit integer addition 255 + 2 might result in 1.

2.5.1.2 Transaction-Ordering Dependence

In blockchain systems, there is no guarantee on the execution order of transactions. 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.

2.5.1.3 Broken Access Control

Access Control issues are common in most systems, not just smart contracts. However, due to the monetary nature and openness of most SCs, properly enforcing access controls are essential. Broken access control can, for example, occur due to wrong visibility settings, giving attackers a relatively straightforward way to access contracts' private assets. However, the bypass methods are sometimes more subtle. For example, in Solidity, reckless use of `delegatecall` in proxy libraries, or the use of the deprecated `tx.origin` might result in broken access control. Code listing 2.1 shows a simple Solidity contract where anyone is able to trigger the contract's self-destruct, which makes the code vulnerable.

Code listing 2.1: Access control vulnerable Solidity Smart Contract code

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

2.5.1.4 Timestamp Dependency

If a Smart Contract is dependent on the timestamp of a transaction, it is vulnerable to attacks. A miner has control over the execution environment for the executing SC. If the SC platform allows for SCs to use the time defined by the execution environment, this can result in a vulnerability. An example vulnerable use is a timestamp used as part of the conditions to perform a critical operation (e.g., sending ether) or as the source of entropy to generate random numbers. Hence, if the miner holds a stake in a contract, he could gain an advantage by choosing a suitable timestamp for a block he is mining. Code listing 2.2 shows an example

Solidity SC code 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 }
```

2.5.1.5 Reentrancy

Reentrancy is a vulnerability that occurs when a SC calls external contracts. Most blockchain platforms that implement SC 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. It is called "reentrancy" since 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 is able to withdraw all the user's funds from a contract. If a malicious actor carefully crafts 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 be eliminated by updating the balance (line 4) before transferring the funds (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  }
```

2.6 Vulnerability detection

Many tools and methods for vulnerability detection have been developed over recent years. This includes both static and dynamic vulnerability techniques, as well as tools based on Machine Learning (ML). These tools can be categorized in terms of their primary function. This includes symbolic execution, syntax analysis, abstract interpretation, data flow analysis, fuzzy testing, and machine learning. In the following sections, the identified vulnerability detection tools are summarized, compared, and analyzed in detail.

Condence into one section, describing the different methods.. Add ontology based detection (for SoliDetector)

2.6.1 Symbolic execution

Symbolic execution is a method for analyzing a computer program in order to determine what inputs cause each part of a program to execute. Symbolic execution requires the program to run. During the execution of the program, symbolic values are used instead of concrete values. The program execution arrives at expressions in terms of symbols for expressions and variables, as well as constraints expressed as symbols for each possible outcome of each conditional branch of the program. Finally, the possible inputs, expressed as symbols, that trigger a branch can be determined by solving the constraints.

2.6.1.1 Syntax analysis

Syntax analysis is a technique for analyzing computer programs by analyzing the syntactical features of a computer program. This usually involves some kind of pattern matching where the source code is first parsed into a tree structure. This tree is then analyzed by looking for vulnerable patterns while traversing the tree.

2.6.1.2 Abstract interpretation

Abstract interpretation is a method to analyze computer programs by soundly approximating the semantics of a computer program. This results in a superset of the concrete program semantics. Normally, this is then used to automatically extract information about the possible executions of computer programs.

2.6.1.3 Data flow analysis

Data flow analysis is a method for analyzing computer programs by gathering information about the flow of data through the source code. This is done by collecting all the possible set of values calculated at different points through a computer program. This method is able to analyze large programs, compared to, for example, symbolic execution.

2.6.1.4 Fuzzy testing

Fuzzing is an automated testing technique for analyzing computer programs. The technique involves supplying invalid, unexpected, or random data inputs to a program in order to uncover bugs. The program is then monitored during execution for unexpected behavior such as crashes, errors, or failing built-in code assertions.

Chapter 3

Related work

This chapter presents related research in the field of source code synthesis. This includes works related to dataset construction, model implementations, model inference guiding, and different approaches to code synthesis. First, various methods for source code synthesis are presented. Then follows a section on various model implementations, and on the generation of datasets, and a section on the generation of models.

3.1 Code synthesis

This section presents some of the various approaches to code synthesis.

Code synthesis is ...

One of the earlier classical works used a probabilistic Probabilistic context-free grammar (PCFG) [10].

Hindle *et al.* [11] 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.* [12] 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 semantics

Programs can also be synthesized by leveraging the semantics of the code. Alon *et al.* [13] 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 [14] builds on the works of Alon *et al.* [13] 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 surrounding context. Pythia by Svyatkovskiy *et al.* [15] 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 Transformers for code synthesis

Inspired by the success of large natural language models such as ELMo, GPT, BERT, XLNet, and RoBERa (CITATION), large-scale Transformer models have been applied in the domains of code synthesis. Feng *et al.* [16] 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 [16], 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. CodeGPT by [17] provides text-to-code generation by training several monolingual GPT-2 transformer models on Python functions and Java methods. For each programming language, one model was pretrained from scratch, while another was fine-tuned on the code corpus, using the standard GPT-2 vocabulary and natural language understanding ability. The Java models was evaluated on the CONCODE dataset, achieving a state-of-the-art performance BLEU score [7] of 28.69 for the model trained from scratch, and 32.79 for the fine-tuned version. Another model version based on GPT-2 is GPT-C by Svyatkovskiy *et al.* [18]. 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 TODO (TOOD) precision of 0.80 and recall of 0.86. 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 8.59 and a TOOD F-score of 24.8 on the CodeSearchNet test set.

citation

cite

The model complexity of transformers has recently sky-rocketed, with model sizes growing to several tens of billions of parameters. GPT-J, a 6 billion parameter model trained on The Pile, an 825GB dataset. 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. 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 . The model solves 28.8% of the problems in the HumanEval dataset , 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 , 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.

cite

cite

cite

Add input, ouputt, meetric , language and model info to table

Table 3.1: Existing language models.

Refs.	Year	Model[a]	Metrics	Languages	Input	Output
[**empty citation**]	**empty citation**	GPT	BLEU	Python	Docstring	Code

[a] Name of the tool or method. If no name exists, a short description or "–" is used.

3.2 Bias in language models

Include security as a bias. Discuss for example gender bias (ex. male vs female jobs) due to datasets.... Same goes with vulnerablilities.. Include stats from gitthub securitty??

Add paper on code security

3.3 Code comment analysis

Docstring analysis 2.3 of https://arxiv.org/pdf/2010.03150.pdf for clustering comments. Uses it to produce different kind of code comments. However, does not use it for function generation.

Try to find papers that reviewe style of comments

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. Section 4.6 describes the project scope. Finally, Section 4.7 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 [19]. 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 [2] fine-tunes GPT-3 [20] on code data from GitHub. The results are impressive. However, these systems still face many problems, especially in regards to different biases, for example, gender and security biases. Because the model is trained on open-source code [2], 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.* [21] found that almost approximately 40% of the generated code by GitHub Copilot is vulnerable. Security flaws in software results yearly in loss of tens of thousands of million dollars. Due to the monetary nature of blockchain, security flaws are even more severe, as exploits of vulnerabilities often directly result in the loss of funds. Further, the immutable nature prevents the possibility of correcting vulnerable code after being deployed. Therefore, the research objective is to develop a system that can generate secure smart contract code automatically, without the need for human intervention. . This thesis

find real number here .

Add comment-aided approach arguments

tries to address the above problems 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 [22]. DSR is typically an iterative process that involves five steps [23]: 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 pro-

posed in this thesis, conveyed as research questions defined in Section 4.2. The following sections; Section 4.4 and Section 4.5 describe a suggestion for how to solve 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.

Check : or ;

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. The first subsection describes how comment The following subsections describe a system that can generate secure smart contract code automatically, this project uses a deep learning approach. Further, for understanding how to put this system to use in a comment-driven approach, a comment analysis is conducted.

4.4.1 Code comments analysis

It is tehrefore needed to analyze the comments in order to investigate how a user can best use this auto-generating system.

Add a description of the code comments analysis. + the differrent types of comments. + different styles?

4.4.2 Language Model

As discussed in section 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 [24] is the largest model that includes code in its pre-training dataset "The Pile", described in Section 4.4.2.4. The research community has found these models to outperform existing open-source GPT systems in qualitative programming evaluations [25]. These findings are further backed by **??**. 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 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 [20], is their best performing model with 175 billion parameters. However, the model is not openly available at the current time. GPT-J [24] 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 [26], 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 4.4.2.4 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 4.1. 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) [27] for position encoding. Opposite to sinusoidal encoding used in standard transformer models (see Section 2.2.2.2), this is shown to result in better model quality in tasks with long text [27].

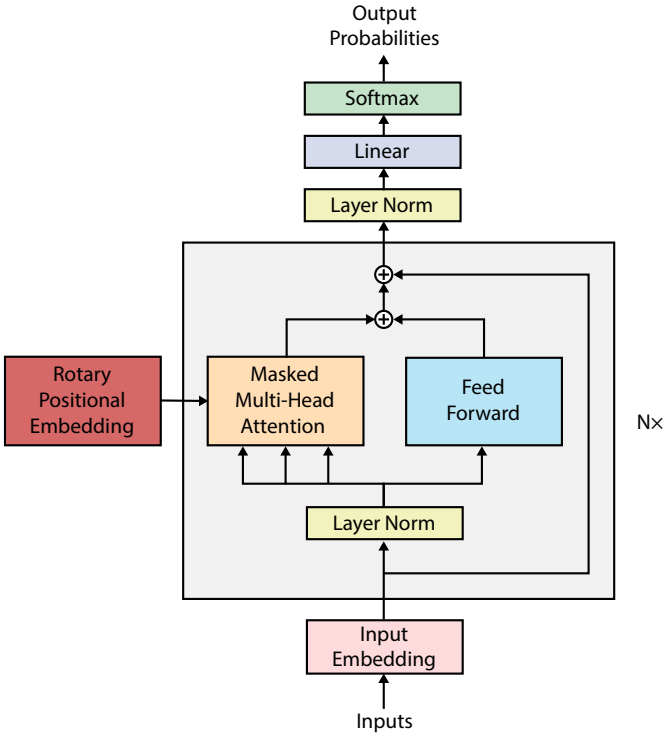


Figure 4.1: Diagram of GPT-J model architecture.

4.4.2.2 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 or 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 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.

4.4.2.3 Pre-training

Pre-training is defined as "Training in advanced". 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 CLM. 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 2.2.3.

4.4.2.4 The Pile

Describe the PILE... It consists of among others, a lot of data from GitHub. However, only x% of the data is smart contracts (Solidity). Hence there is a need for a dataset made up of smart contracts. –> existing datasets....

4.4.3 Metrics

For evaluating the model performance, the accuracy and perplexity metrics are used. For evaluating the function genertion from comments, the bleu score will be used. See .

Do i need this section?

4.5 Design for RQ2

Use secure vs vulnerabile as input to the model inorder to generate secure code. Why might this idea work? Justify.

4.5.1 Security Conditioning

Rewrite - this is a draft

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. In the case of Smart Contracts, the vulnerability analysis presented in section 5.1.1.2 shows that almost 50% of deployed Smart Contracts contain

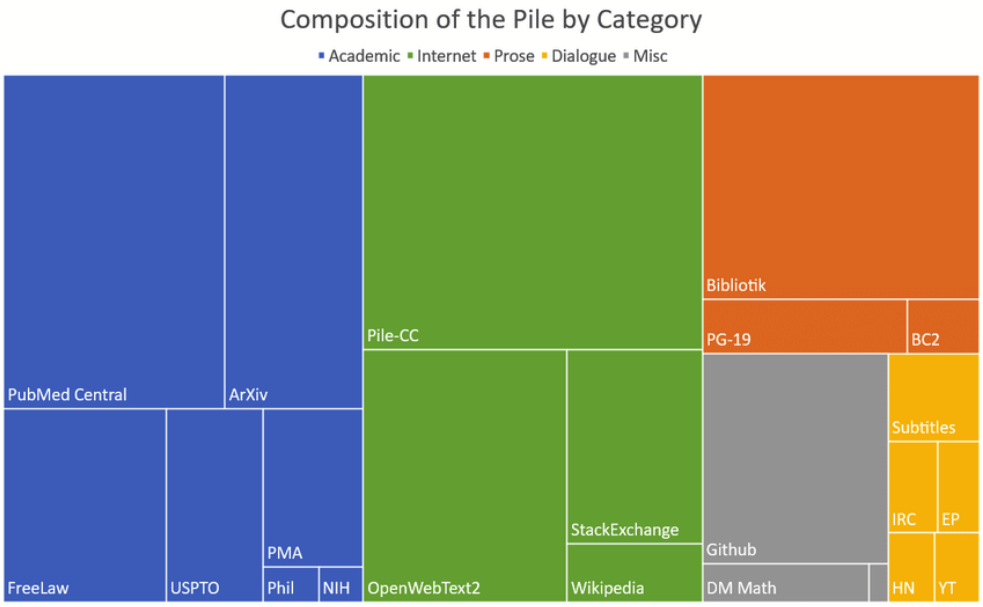


Figure 4.2: Treemap of Pile components by effective size. SOURCE FROM THEPILE paper

at least one high-severity vulnerability. This will result in a biased model that may produce a lot of vulnerable code. This section introduces a technique, named security conditioning, to reduce and mitigate this problem.

Vulnerability analysis is a difficult area. It is especially hard in the area of smart contracts, where the execution environment is not deterministic ????... Previous works have tried to classify vulnerable code with large language models without much success . In this project, instead of classifying vulnerable code, the goal is to make the model more secure by conditioning it on the presence of vulnerabilities.

find correct wording.

The security conditioning is done by appending a special security label to each of the records in the training data. This way, the model can use this token(s) to condition whether to produce safe or vulnerable code. This requires the dataset to first be labeled as secure or vulnerable. For this project, SolidityDetector is used for labeling. Further details on the dataset construction can be found in Section 5.2.1.2.

Cite previous works

4.5.2 Metrics

For evaluating the model has not droppedin performance, the accuracy and perplexity metrics are used.

Do i need this section?

4.6 Project scope

The project scope in this thesis is limited to the generation of smart contracts for the Ethereum blockchain. Further, only one language model architecture is used. Specifically, the state-of-the-art open-sourced pre-trained transformer model GPT-J-6B by ElutherAI is selected. For research questions 1 and 2, two versions of the model were created by fine-tuning it on two smart contract datasets created for this project. Due to the share size of this model (see Section 4.4.2.2), no hyper-parameter optimization was performed. The hyper-parameters were left to defaults used during pre-training. Hence, everything but the training data is kept constant throughout the experiments. For research question 3, the primary scope is to assess variations in the actual input to the model. Thus, differences in performance due to variations of the models' inference configuration are not thoroughly explored.

Several options are available for tuning the code generation, including *temperature*, *stops*, and *top p*.

Restricted to mainly Solidity language

4.7 Technology

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

4.7.1 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 [28] library by Hugging Face [29] fulfilled these conditions. The library provides flexible and easy-to-use solutions. It also supports integration with DeepSpeed [30], a deep learning optimization library by Microsoft [31] 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.

NLTK, extensive use of Pandas, as well as Pytorch, Numpy, and Matplotlib are used for data processing and visualization. For traching training experiments, WANDB is used.

Fix

DeepSpeed. The deep learning optimization library DeepSpeed [30] 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) [32]. 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 [33] 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 [33]. However, each optimization stage comes with a performance cost, slowing down the training process. DeepSpeed also provides support for mixed-precision training [34]. 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.

4.7.2 Hardware resources

IDUN High Performance Computing Platforms [35]. IDUN full-fills the requirements defined in Section 4.4.2.2.

Describe more in detail

Figure 4.3: Image of IDUN todo: add ref `https://www.hpc.ntnu.no/idun/`

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 scraped verified 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

`https://github.com/andstor/smart-contract-downloader`

The largest provider of verified SCs is Etherscan. This website provides a list of all verified SCs on the blockchain. More on their service..... Etherscan provides a API for downloading verified Smart Contracts. The API is available at `https://api.etherscan.io/api`.

In order to download the SCs from Etherscan, a tool we need to provide the SCs address. The address is the first part of the SCs code. The address is the first part of the SCs code.

The following code snippet is a Google BigQuery query. It will select 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, and is 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
}
```

Saved to file for simple restarting, multiprocessing and parallelization.

The total number of files generated by the downloading program 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.

Include img of the processing script output

Normalization. The most common is a contract written the Solidity language with a single contract "entry" . However, a single contrract file can contain multtiple contracts, making use of properties like inheritance etc.. The source code contracts can also be split over multiple files, a formmat rreefered to as "Multi file". When compiling thtese, the source code files aree "flattened" into a single contract file before compiliattion. Anotther flavour is hte JSON format, which is a language that is used to describe the SCs. Here the sourcecode is structured in tthe in the JSON code. Smart contracts can also be vritten in the Vyper language. Vyper is .... .

Find a better name for contract keyword

explain vyper

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.

Duplication filtering. A large quantity of Smart Contracts contains duplicated code. Primarily, this is due to the frequent use of library code, such as Safemath and ... . 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 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 discssed in detail in the following sections.

Reference libraries

The actual code filtering is done by applying a token-based similarity algorithm named Jacard Index. The algorithm is computationally efficient and can be used to filter out SCs that are not similar to the query. The Jacard Index is a measure of the similarity between two sets. The Jacard Index is defined as the ratio of the size of the intersection to the size of the union of the two sets.

5.1.1.2 Verified Smart Contracts

`https://huggingface.co/datasets/andstor/smart_contracts` Verified Smart Contracts is a dataset of real Ethereum Smart Contract, containing both Solidity and Vyper source code. 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 blockchain. A set of 100,000 to 200,000 contracts are provided, containing both Solidity and Vyper code. In the following paragraphs, we will discuss the different dataset versions in detail. It consists of every deployed Ethereum Smart Contract as of 1st of April 2022, whose been verified on Etheres-can and has a least one transaction. The dataset is available on request at `https://huggingface.co/datasets/andstor/verified-smart-contracts`. Table 5.1 shows the metrics of the various (sub)datasets.

All processing scripts are available at `https://github.com/andstor/verified-s` arsed.

Remove parsed from table Table 5.1

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
Parsed	4.44 GiB	4,434,014	29,965,185

LoC refers to the lines of source_code. The Parsed dataset counts lines of func_code + func_documentation.

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.

Add stats on the raw dataset

Flattened. The flattened dataset is a filtered version of the Raw datasetSection 5.1.1.2. 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. 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.

The following command prooduces the flattened dataset:

```
python script/filter_data.py -s parquet -o data/flattened --threshold 0.9
```

Code listing 5.3: Solidity standard JSON Input format.

```
{
  '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'
}
```

Inflated. The inflated dataset is also based on the raw dataset. Each contract file in the dataset is split into its original representative files. This mitigates a lot of the problems of the flattened dataset in terms of duplicated library code. The library code would, along with other imported contract files, be split 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 dataset. The dataset is then split 80%, 10%, 10% into a training, validation and test set, respectively.

```
    python script/filter_data.py -s parquet -o data/inflated --split-files --threshold
0.9 dupes=5217191/5403136 (96.56
```

Code listing 5.4: Solidity standard JSON Input format.

```
    {
        '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...',
7       'abi': '[{"inputs":[],"stateMutability":"nonpayable","type":"constructor"}
    ↪ ...]',
8       'compiler_version': 'v0.8.4+commit.c7e474f2',
9       'optimization_used': False,
10      'runs': 200,
11      'constructor_arguments': '',
12      'evm_version': 'Default',
13      'library': '',
14      'license_type': 'MIT',
15      'proxy': False,
16      'implementation': '',
17      'swarm_source': 'ipfs://eb0ac9491a04e7a196280fd27ce355a85d79b34c7b0a83ab606
    ↪ d27972a06050c'
18    }
```

Plain text. For easy use of the dataset for casual language modeling training, 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.

smart_contracts_dataset :: inflated_plain_text tokens: test: 74314082 validation: 67064318 train: 687186308 TOTAL: 828.564.708

5.1.2 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 the 6, shedding some light on which commenting style is the best to use.

5.1.2.1 Universal Solidity parser

For being able 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 [36]. ANTLR4 [37] 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 [38]. ANTLR is primarily a Java application. However, several code generation targets are available, including Java, C#, Python, JavaScript, Go, C++, Swift, PHP and Dart [39]. 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 [40] for the official Solidity grammar [36] 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.

5.1.2.2 Verified Smart Contract Code Comments

`https://huggingface.co/datasets/andstor/smart_contract_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 [41] 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 .... Code listing 5.5 shows an example data instance from the dataset.

Add stats for comments dataset

Code listing 5.5: Solidity standard JSON Input format.

```
{
    '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...',
```

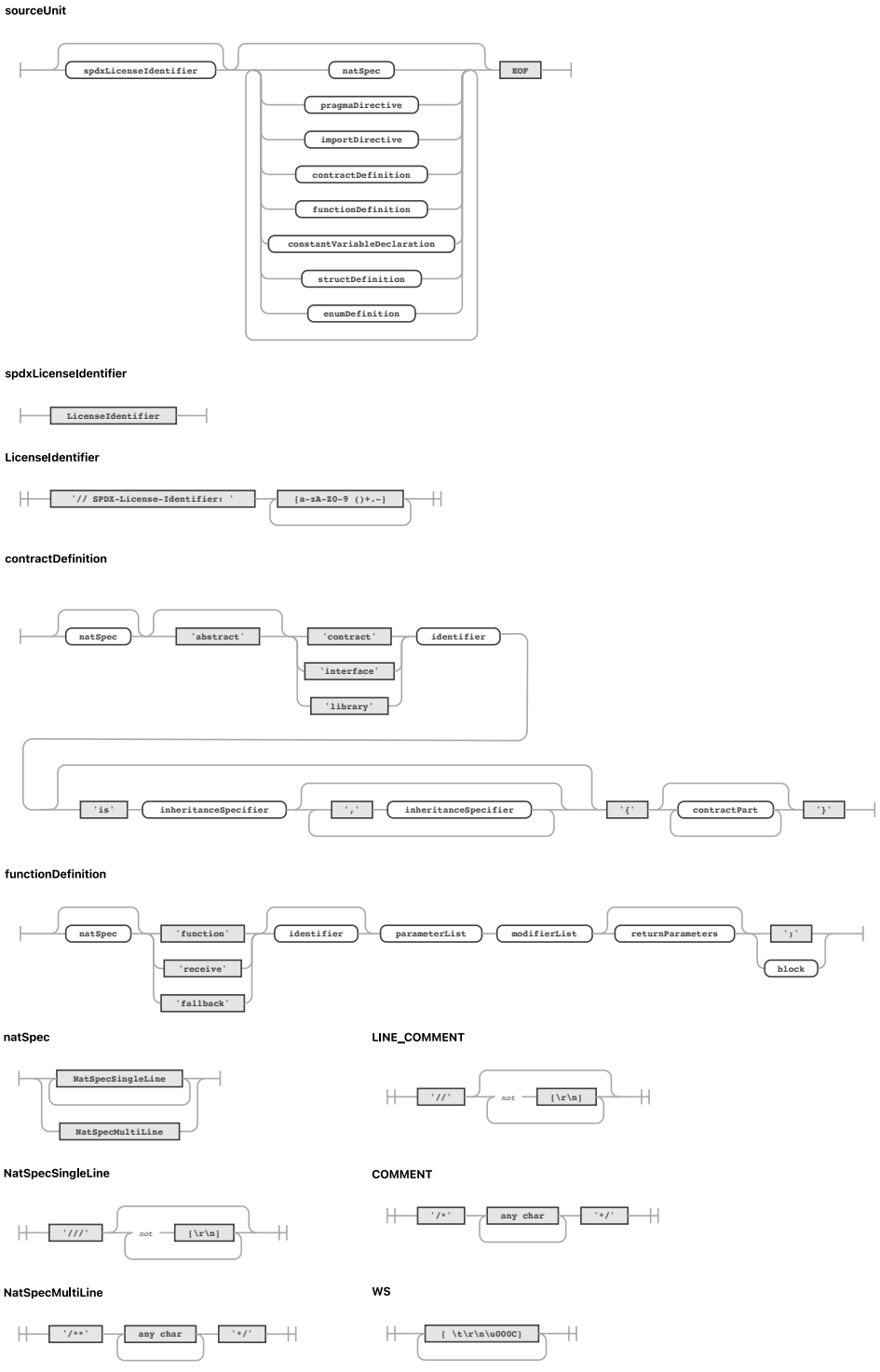


Figure 5.1: Railroad diagrams of main code comment alteration to Solidity grammar.

```
8     'class_documentation': '/// @title Stake Delegatable\n/// @notice A base
      ↪ contract to allow stake delegation for staking contracts.',
9     'class_documentation_type': 'NatSpecSingleLine',
10    'func_name': 'balanceOf',
11    'func_code': 'function balanceOf(address _address) public view returns (uint256
      ↪  balance) {\n        return operators[_address].packedParams.getAmount();\n
      ↪     }',
12    '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.',
13    'func_documentation_type': 'NatSpecSingleLine',
14    'compiler_version': 'v0.5.17+commit.d19bba13',
15    'license_type': 'MIT',
16    'swarm_source': 'bzzr://63a152bdeccda501f3e5b77f97918c5500bb7ae07637beba7fae76
      ↪ dbe818bda4'
17 }
```

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 NLTK (Natural Language Toolkit) [**nltk**] 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 [42] 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 [43] 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 Term Frequency–Inverse Document Frequency (TFIDF) (see Section 2.1.2.2) 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 100-dimensional vectors are reduced to 2D using Principal Component Analysis (PCA). The clustering result is shown in Figure 5.4. The explained variance captured in the 2D plot is approximately 0.64, as shown in the Scree Plot in Figure 5.3.

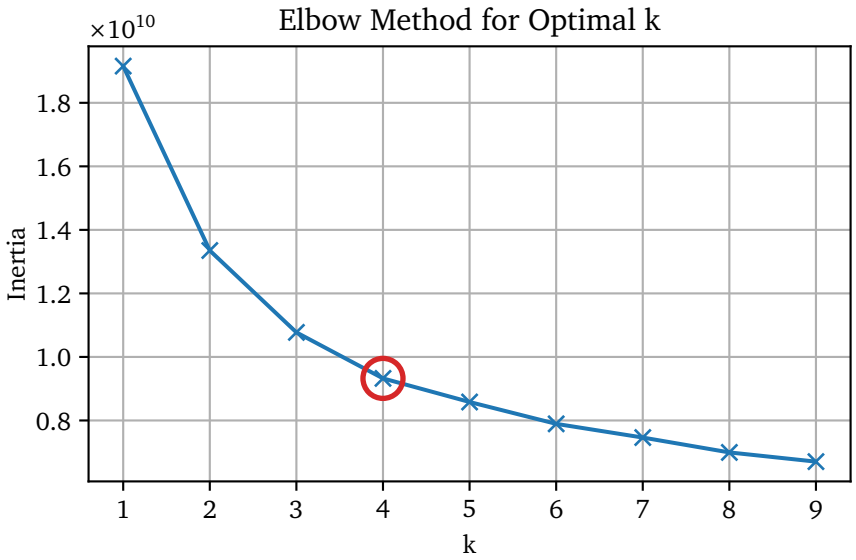


Figure 5.2: Elbow method for determining the optimal number of clusters.

Code listings 5.6 to 5.9 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 5.6. 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 5.8 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 [44], 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 5.9 marks the start and end of the comments with many dashes.

Code listing 5.6: NatSpec single-line comment in cluster 0.

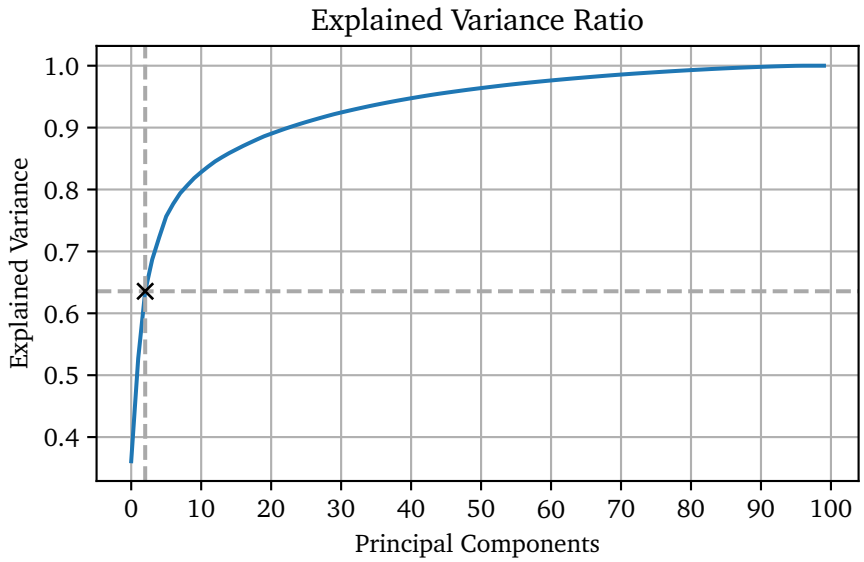


Figure 5.3: Scree Plot for the PCA dimensionality reduction

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

Code listing 5.7: Single-line comment in cluster 1.

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

Code listing 5.8: NatSpec multi-line comment in cluster 2.

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

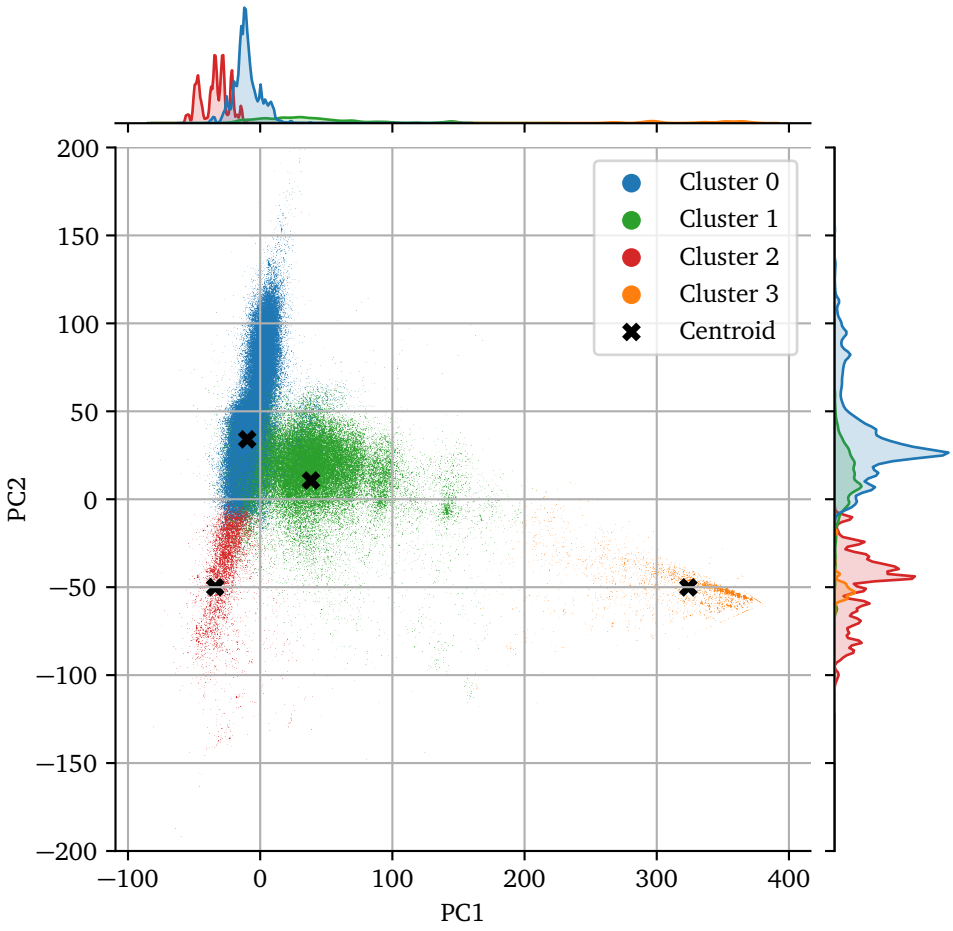


Figure 5.4: 2D plot of the comment clusters.

Code listing 5.9: 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.

Add figure of training process?

5.1.3.1 Pre-training

In this project, pre-trained weights for GPT-J-6B from ElutherAI are used. See Section 4.4.2.1 for a description of the model architecture. The pre-training by ElutherAI is done on the dataset The Pile, described in Section 4.4.2.4. 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.

add table notes - * each layer consists of one feedforward block and one

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 2.2.3. 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 [30] is used as a wrapper around the HuggingFace library. See Section 4.7.1 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 5.6 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 5.10. 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 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.

[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 5.10: 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
```

Fix deep-speed table.. Eg. cpu offload-ing... Use json config...

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
offload_param	null
offload_optimizer	device: null
	nvme_path: null
	buffer_count: 4
	pin_memory: false
	pipeline_read: false
	pipeline_write: false
	fast_init: false
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

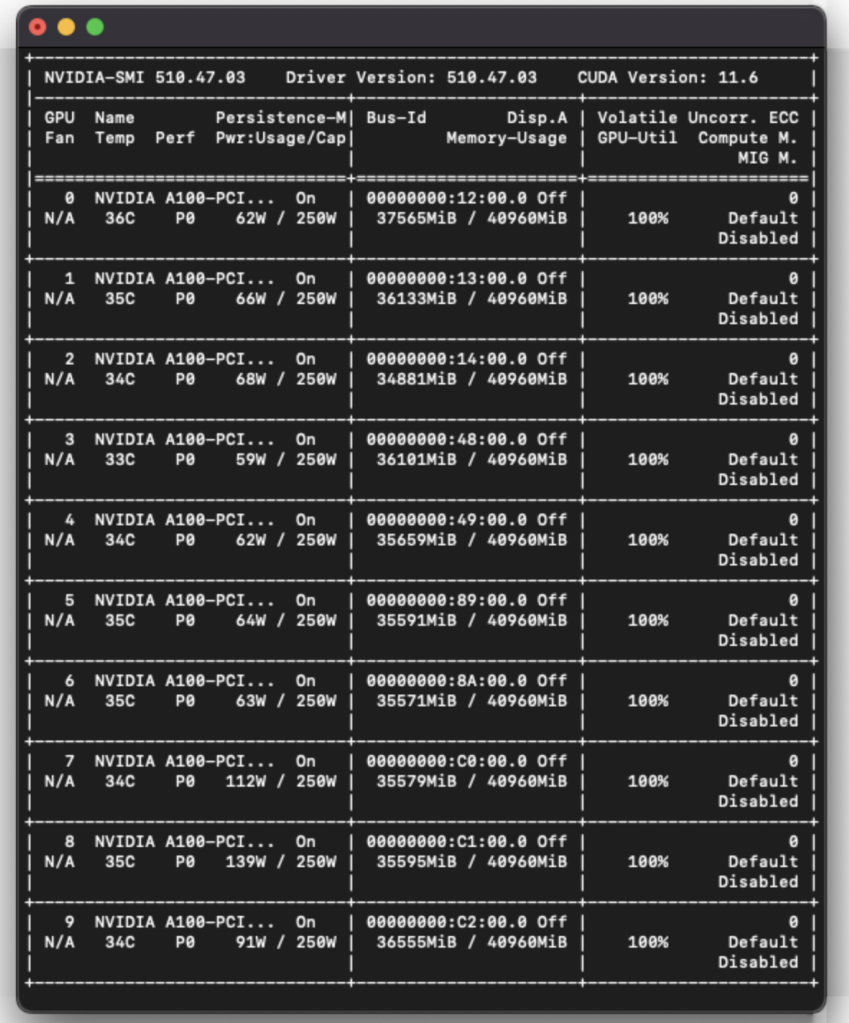


Figure 5.5: Screenshot of nvidia-smi program showing 100% GPU utilization.

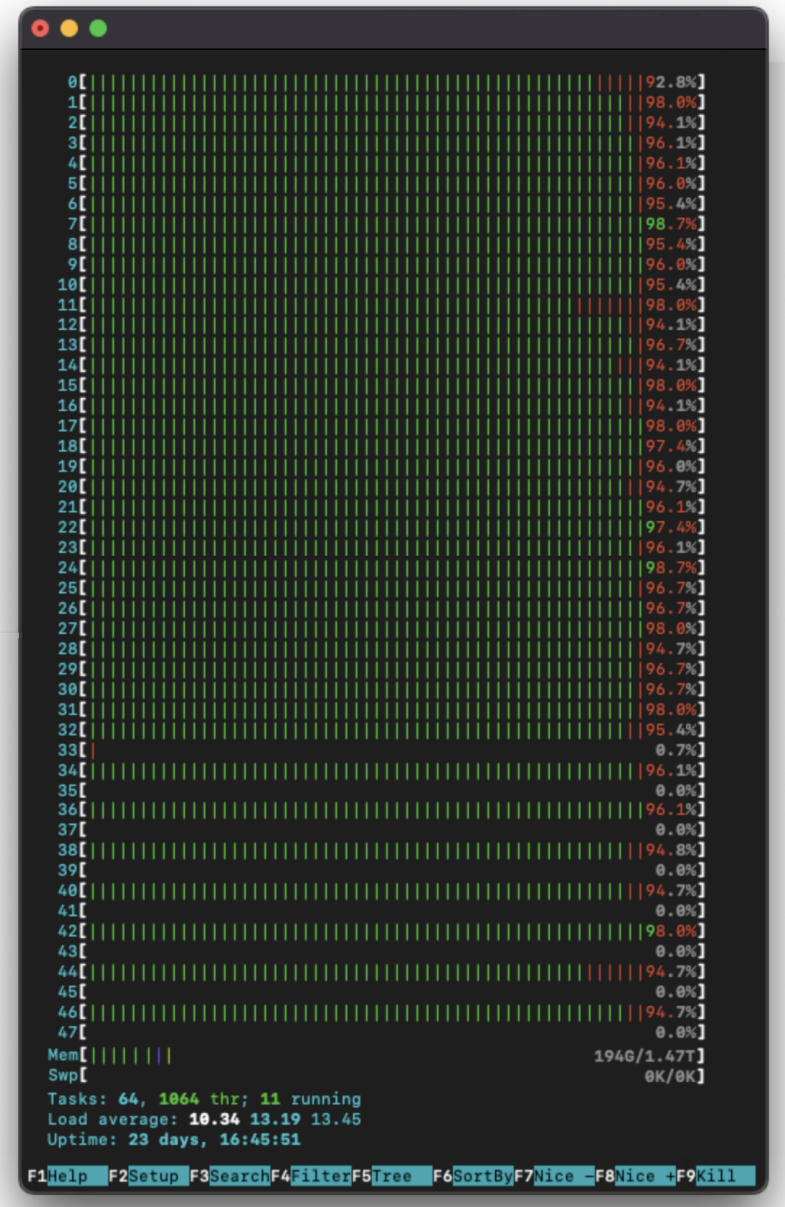


Figure 5.6: Screenshot of htop program showing host CPU and memory activity during optimizer computation.

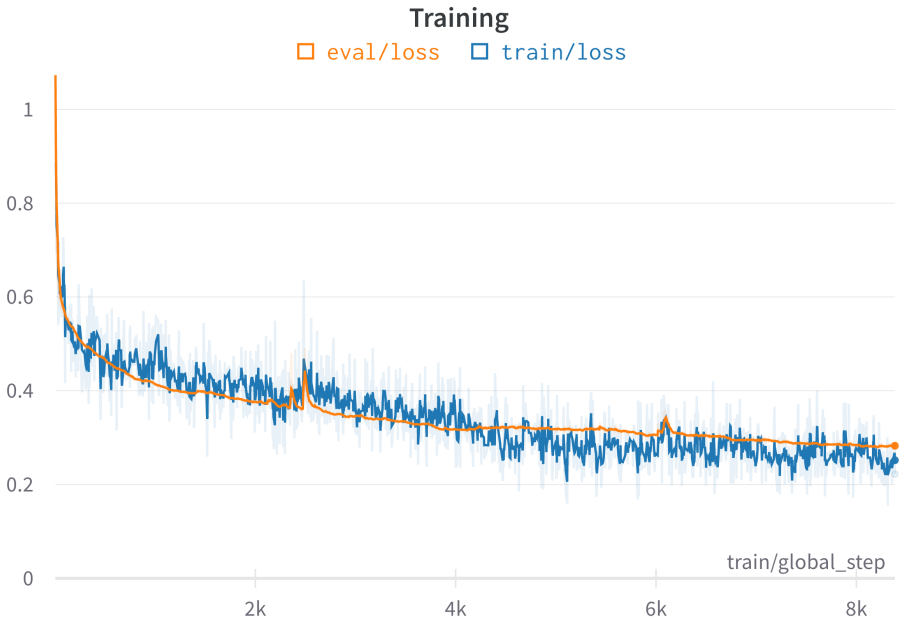


Figure 5.7: Training and evaluation loss during model training.

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 technique is rather easy, as 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 SoliDetector is used. The choice of using SoliDetector is due for two reasons. Firstly, as SoliDetector 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 that for example use symbolic analysis would only work on the flattened dataset version (see Section 5.1.1.2). Secondly, SoliDetector works with any Solidity version.

What vulnerabilities can tthis detect? ++ make table

SoliDetector takes in a SC file and outputs the vulnerability analysis results as a text file. However, due to the large number of contracts needed to be labeled in this project, SoliDetector is run in parallel. A python script is created that leverages multiprocessing to run SoliDetector in parallel. Since SoliDetector is a Java program, it is run as a child process and controlled with the help of the python module Pexpect [45]. 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

The results of the vulnerability labeling are packed into a dataset named Verified Smart Contracts Audit. This is done for both the flattened and inflated dataset versions. The finished dataset is available at `https://huggingface.co/datasets/andstor/smart_contracts_audit`. Both dataset versions keep the original split

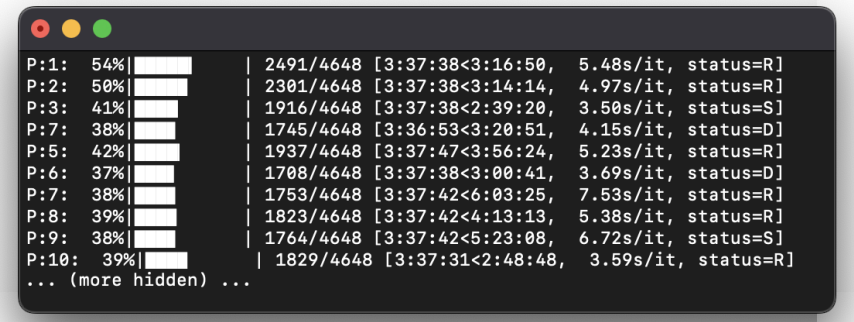


Figure 5.9: Screenshot from the vulnerability labeling process with SoliDetector.

into a training, validation and test set (80%, 10%, 10%). Code listing 5.11 shows an example data instance from the audited flattened dataset.

smart_contracts_dataset-audit :: inflated_solidetector_embedded tokens: test: 74332721 validation: 67082958 train: 687335426 TOTAL: 828.751.105

Code listing 5.11: Solidity standard JSON Input format.

```
{
    '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).

Add example listing of security conditioning.

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

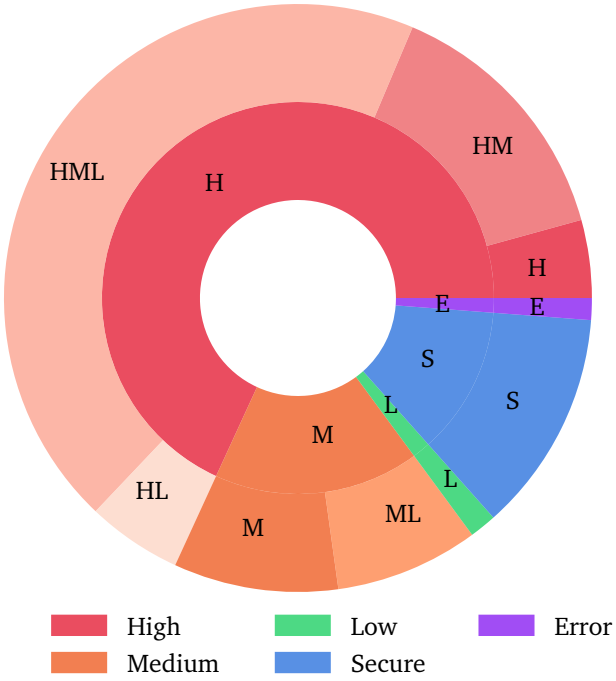


Figure 5.10: Doughnut chart over the distribution of the vulnerability severities in the flattened dataset at different granularity levels, where each level occurs at least once in the SC.

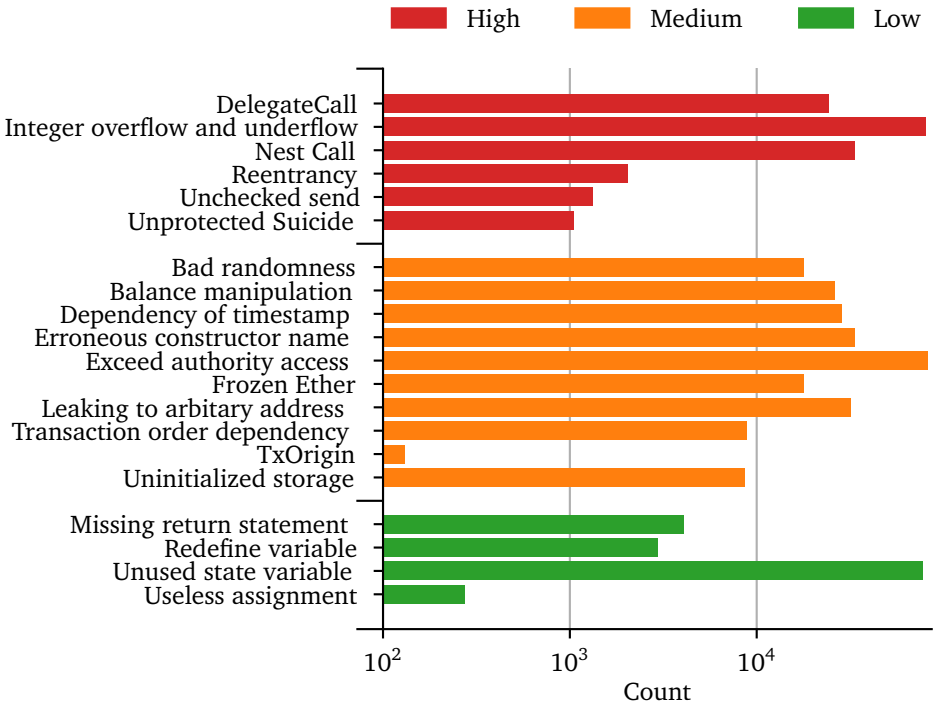


Figure 5.11: Distribution of vulnerabilities in the flattened dataset.

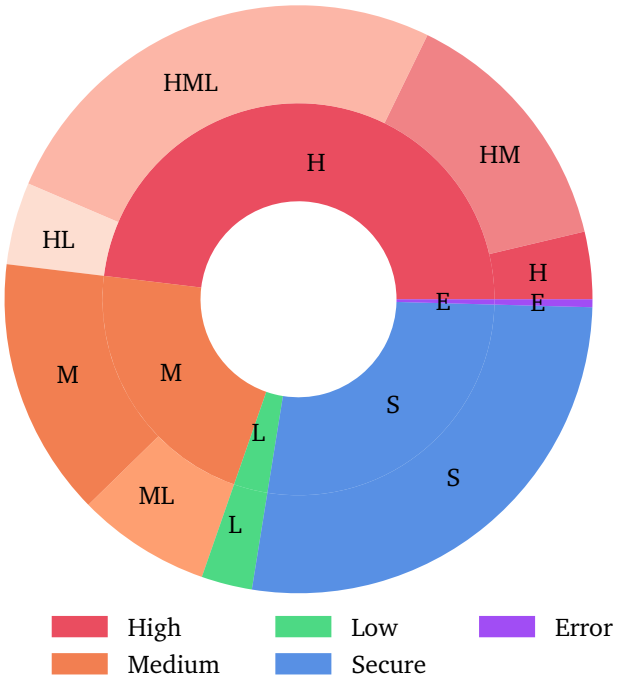


Figure 5.12: Doughnut chart over the distribution of the vulnerability severities in the inflated dataset at different granularity levels, where each level occurs at least once in the SC.

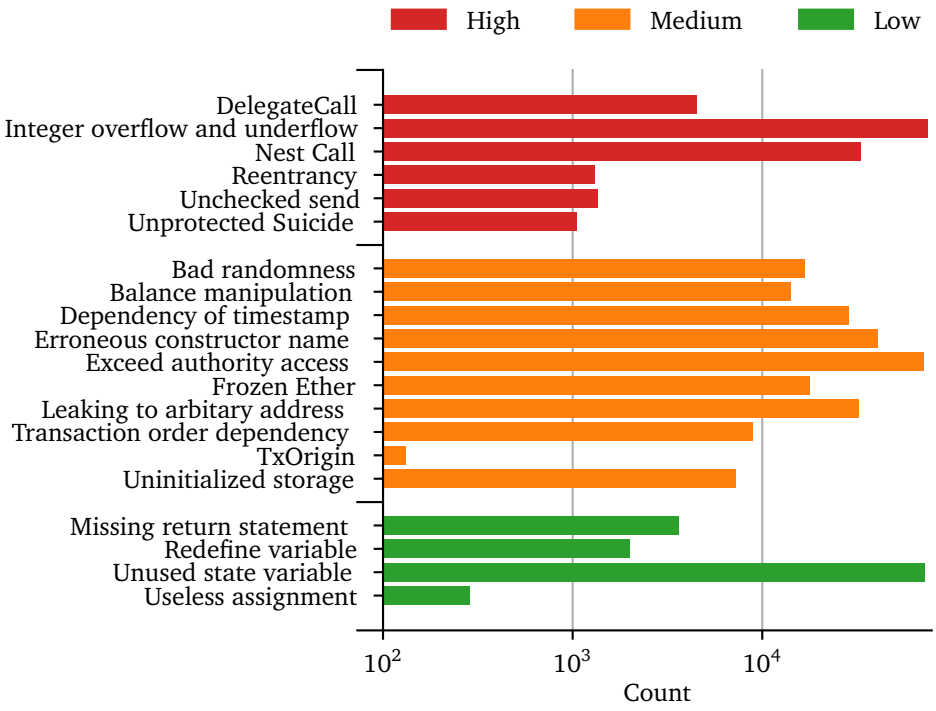


Figure 5.13: Distribution of vulnerabilities in the inflated dataset.

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 1.511. 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.

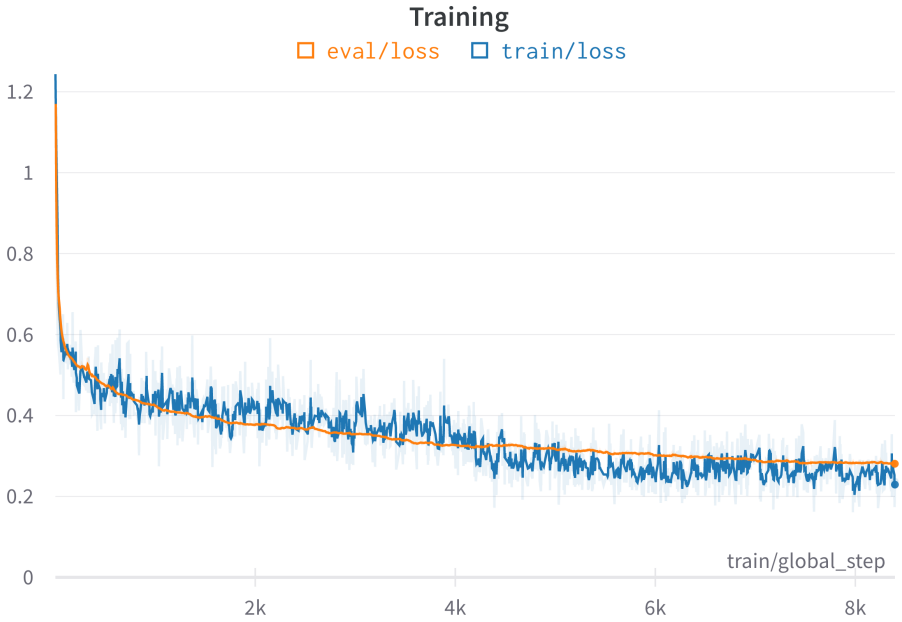


Figure 5.14: Training and evaluation loss during training of model with security conditioning.

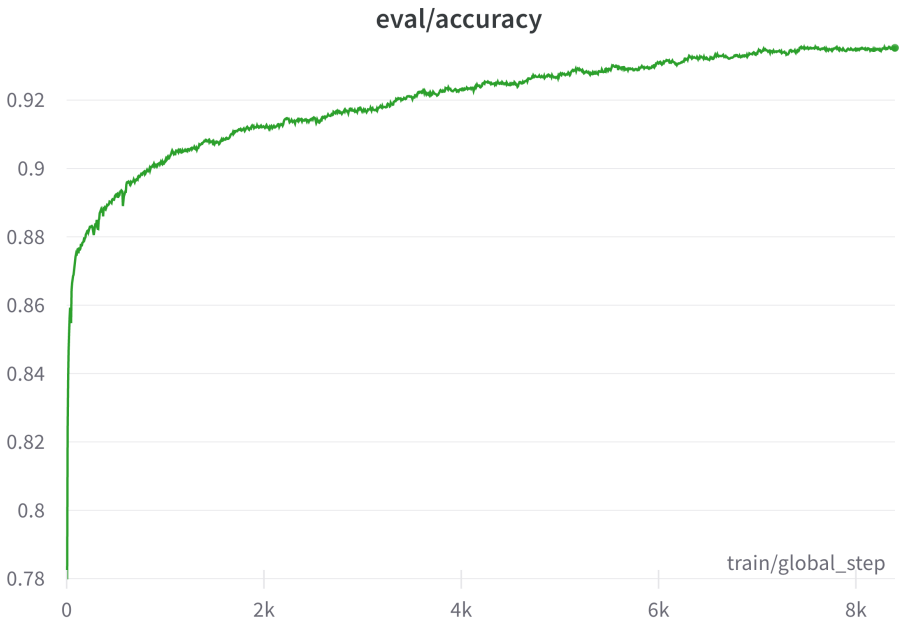


Figure 5.15: Evaluation plot of accuracy during training of model with security conditioning.

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 then evaluate two scenarios. One

6.1.1 Evaluation Method

The evaluation strategy employed is to measure the similarity between generated code and original code. 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.

This evaluation strategy is sound, as it captures how such a system would perform in real life.

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. **??** runs an evaluation utilizing the maximum amount of code context available, and Section 6.1.4 runs an evaluation using only function comments as input. Since the model is auto-regressive, a custom stopping strategy based on matching braces is implemented for generating well-formed functions.

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 **??**. BLEU is a commonly used evaluation metric within the area of code synthesis [46]. 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 [46]. Because of this, adaptations such as CodeBLEU by Ren *et al.* [46] 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, 47] 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 + 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 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, and the BLEU score is calculated by comparing it against the actual function. This evaluation procedure is done for both the pre-trained model and the fine-tuned model.

Figures 6.1 and 6.2 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 6.1) 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,

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

the pre-trained model does not performe well for generating SCs. The results from the fine-tuned model (Figure 6.2)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.

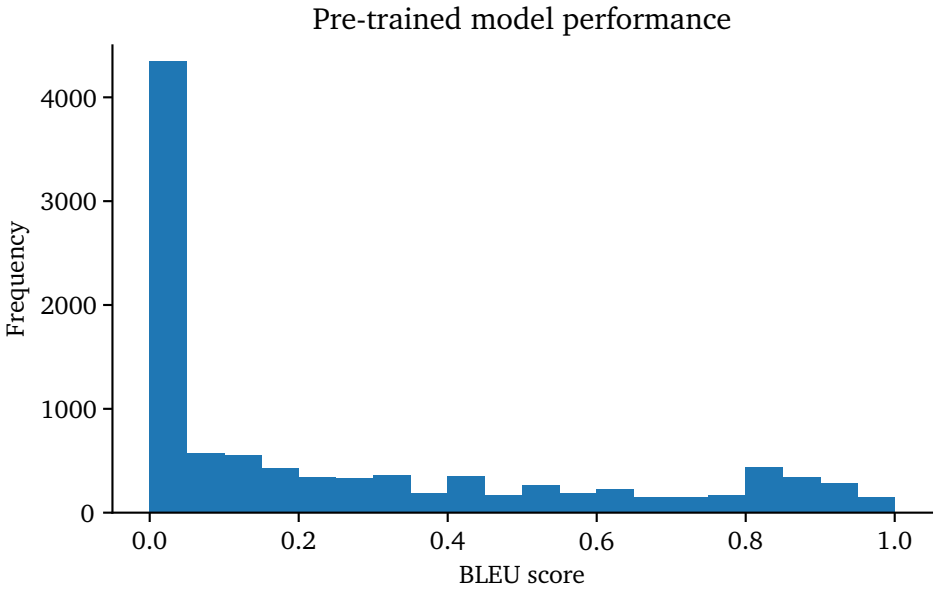


Figure 6.1: BLEU score frequency distribution of 10.000 generated functions with pre-trained model using full code context.

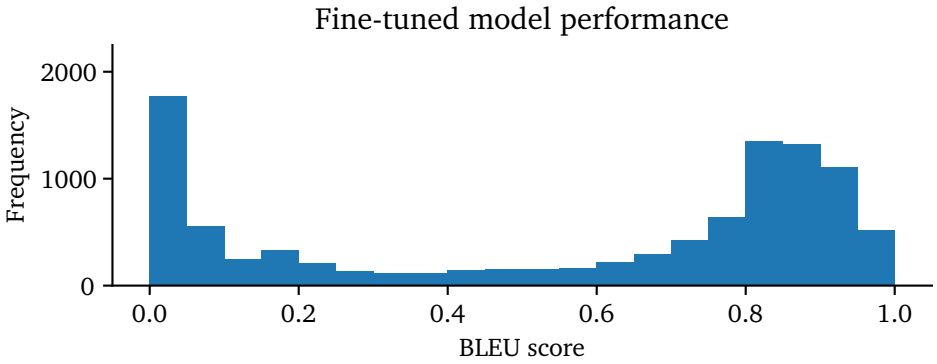


Figure 6.2: BLEU score frequency distribution of 10.000 generated functions with fine-tuned model using function comments + all available code context.

6.1.4 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 **??**. 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, the function comment plus the function signature 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. The choice of adding a function signature is done to make the comparison between different clusters fair, as well as account for some of the problems with the BLEU score. For example, even though the function is functionally correct BLEU penalizes wrong variable names. By providing parameters to the model, some of these problems are reduced.

?? 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. From the plots, it is clear that the fine-tuned model performs significantly better than the pre-trained model. The performance for the pre-trained model is ranked from worst to best as follows: cluster 1, cluster 0, cluster 3 and cluster 2. The averaged BLEU scores can be seen in Table 6.1.

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 the function parameters (see Code listing 5.6). Since this evaluation run adds function signatures to the comments, a lot of the information in the comments of cluster 0 is redundant. The main difference is the actual description of the parameters. Still, the description of the parameters results in about a 60% increase of the BLEU score. However, both cluster 2 and cluster 3 significantly outperform cluster 2, as can be seen from Table 6.1. As discovered in Section 5.1.2.3, cluster 2 contain a lot of library code implementations. It is therefore reasonable to assume that the model excels in generating code for the implementation of popular libraries. Cluster 3 is the best performing of all the clusters. However, it presents a rather interesting distribution with some large peaks to the far right. Upon manual inspection, most of the outliers are part of a popular ERC20 [2] token implementation from an old tutorial [48] from 2017. This also explains the outliers in 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 4.4.2.4).

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

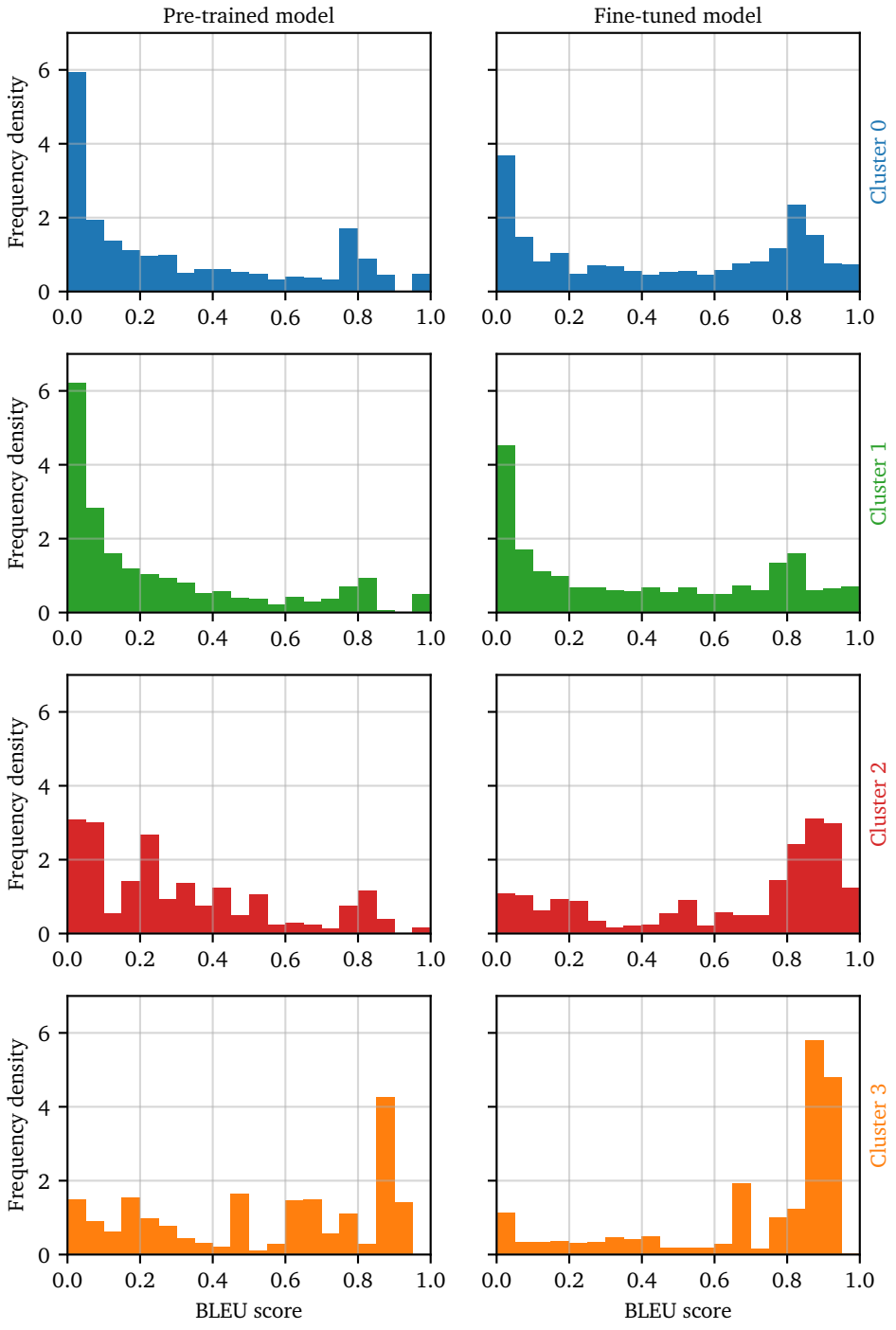


Figure 6.3: BLEU score frequency distribution of comment clusters using comment + function identifier.

Table 6.1: BLEU score of only comment generation.

Pre-trained model	Cluster 0	Cluster 1	Cluster 2	Cluster 3
Pre-trained model	0.303	0.249	0.303	0.533
Fine-tuned model	0.456	0.391	0.631	0.696

6.1.5 Transfer learning

Until now, the focus has been on Solidity. However, the training data also contains a very small amount of Vyper code. example

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. The evaluation method used for this evaluation is described in detail in Section 6.2.1. An evaluation dataset is used for some of the evaluation runs are then presented in **??**. Finally, the results from the different evaluation runs are shown in **??**.

6.2.1 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:

1. Some code from a 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 original input is prepended to both the generated output and the target value
5. The two prepended code parts are fed into a vulnerability detection tool.
6. The number of vulnerabilities from the two analyses are counted and compared.

WRONG!!

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. For the model with security conditioning, two rounds oof generation are performed. One round with prepending the security label "<|secure|>", and one round with "<|vulnerable|>". The three results are then run through SoliDetector for vulnerability analysis, and the results are compared.

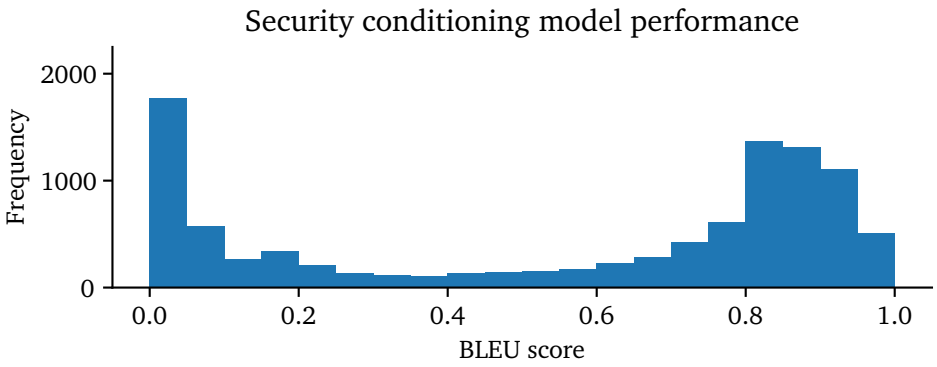


Figure 6.4: BLEU score frequency distribution of comment clusters.

Section 6.2.2 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). However, this method of evaluation does have some drawbacks when used on existing real contracts. As discovered in Section 5.2.1.2, almost 50% of the contracts used for evaluation contains 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. The obvious choice to mitigate this would be to only use comments as input for code generation. However, the generated output would not be possible to analyze automatically. For example, the generated function might add a number to a class state variable, introducing an integer overflow vulnerability. While the model is able to "guess" the existence of such a state variable, a vulnerability detection tool would normally not label this as a vulnerability. Instead, this would be considered by most tools as a useless assignment, as it is technically not a vulnerability - yet. To mitigate this, a custom evaluation dataset is created (see **??**) and used for evaluation in Section 6.2.4.

6.2.2 Evaluation using real smart contracts

Figure 6.5

Describe figure 6.5

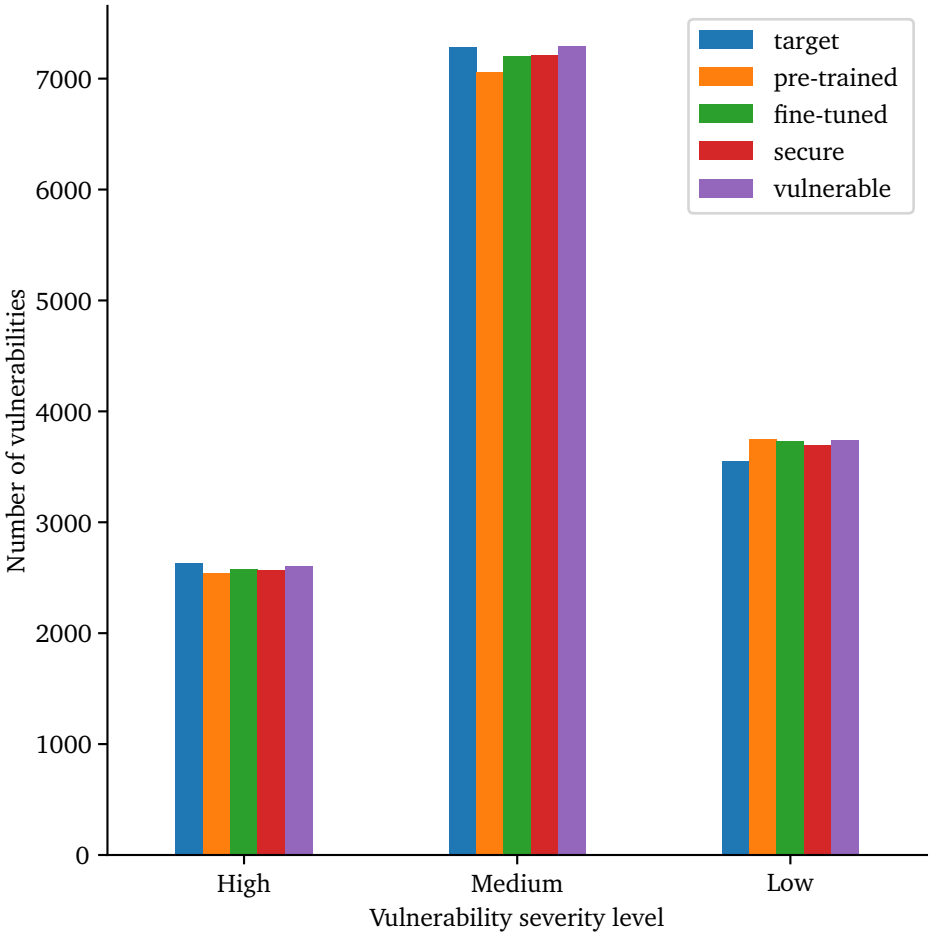


Figure 6.5: Count of vulnerabilities.

6.2.3 Prone Contracts Dataset

Custom dataset containing multiple hand written INCOMPLETE contracts that MAY produce vulnerabilities.

Make example dataset.

6.2.4 Evaluation using Prone Contracts

By using a custom made evaluation dataset, it can be hypotthesized that this more accurately capttures the behaviour of the system as it would be used in production.

Using security conditioning reduces the frequency of vulnerable code.

Show example dataset results.

6.2.5 Model weigts

Can we find structures in the weights? Neural view? bertviz? That resembles AST equivalent? Answers to research questions Evaluation of the answers Get logits from a model prediction to visualize the distribution of the predicted probabilities.

Add this if enough time

Chapter 7

Discussion

In this chapter, the results of the implementation and evaluation given in Chapters 5 and 6 are discussed. In **??** various threats to validity are discussed.

7.1 Comparison with related work

This section compares the results and findings with related work. First, the transformer model fine-tuned fine-tuned for SC code generation is discussed. Then, the security conditioning approach developed for answering research question 2 is discussed.

Remove the "Discussion of RQx" headings?s

7.1.1 Discussion of RQ1

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. 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.

Try to find another work fine-tuning on programming language and reporting accuracy+perplexity improvement

This work also considers evaluation using a comment-aided approach for generating code. In Section 6.1.3, the model is evaluated using comments and code context.

In Chapter 6, the model is evaluated on method generation from comments. While most other works related to automatic function generation from comments, few have looked at extending this together with code context. As a user seldom has to create code without the combining specific combination of comments and combined the primarily been interested in the performance of . Few works take into account how the model is used by the user. As doing user studies are a verry

time consuming and difficult procedure, very few works have been able to answer the research question. The implementation of this thesis has been able to answer the research question.

The model is evakluated for function generation from code comments. Due to the lack of TThis is the first model purposed for generating smart contract code.

https://minimaxir.com/2021/06/gpt-j-6b/

For training such a large model, this project constructs the largest real SC dataset ever created. Alternative

, and a state-of-the-art automatic smart contract code generation model.

There have been several Research question 1 is to implement a transformer model for generating SC code automatically, using a comment-audeed approach. Compared to related ... Compare BLEU score here.. make table!! Small sttattement regarding the paper stating 40% vulnerable code from copilot.

7.1.2 Discussion of RQ2

Too answer research question 2, this thesis has investigated how to automatically generate Smart Contract code with transformer-based language models

7.2 Threats to Validity

Data contamination in test and train datasets.

Only use one vulnerability detection tool

Chapter 8

Future work

The area of SC vulnerability analysis and detection has already come a long way, even though the area of blockchain is still in its infancy. There are still many research gaps needing to be filled.

Use the model itself for clustering.

train model from scratch on only smart contract code.. Not possible due to time and resource requirements.

Reduce model size with knowledge distillation

Faults in SoliDettector...

Future work, combine multiple vulnerability detectors

The area of vulnerability evaluation of code synthesis is

An efficient and sound approach for automatically evaluating the security of synthesized code is lacking. Current sound solutions require a lot of manual effort. A custom vulnerability detection tool that could detect not just definite vulnerabilities, but also potential vulnerabilities could be an interesting solution and is left as future work.

Chapter 9

Conclusion

Consectetur ullamco dolor pariatur ad id minim sunt do occaecat. Anim commodo consectetur proident pariatur dolor. Dolor laborum nisi id ipsum eiusmod ipsum exercitation consequat ullamco pariatur ex ut ullamco id.

Write a conclusion

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] 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.

[8] L. S. Sankar, M. Sindhu, and M. Sethumadhavan, "Survey of consensus protocols on blockchain applications," in *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2017, pp. 1–5. DOI: 10.1109/ICACCS.2017.8014672.

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

[10] 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/.

[11] 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.

[12] 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/.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, *Codexglue: A machine learning benchmark dataset for code understanding and generation*, 2021. DOI: 10.48550/ARXIV.2102.04664. [Online]. Available: https://arxiv.org/abs/2102.04664.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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/.

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

[25] 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/.

[26] ElutherAI. "Elutherai." (Apr. 2022), [Online]. Available: https://www.eleuther.ai (visited on 06/28/2022).

[27] 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.

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

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

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

[38] 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.

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

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

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

[42] 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.

[43] 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.

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

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

[46] 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.

[47] 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.

[48] 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).

André Storhaug

Secure Smart Contract Code Synthesis

NTNU
Norwegian University of
Science and Technology