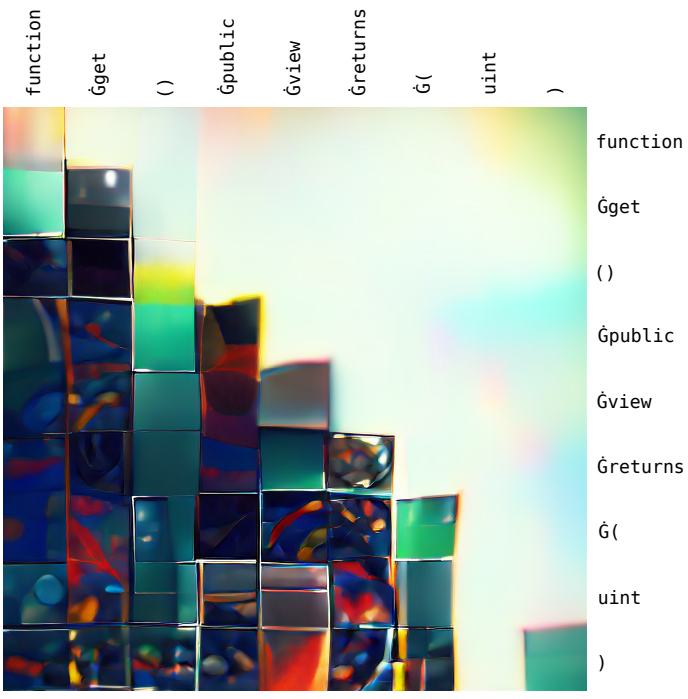
Master's thesis

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

Vulnerability detection and security of Smart Contracts are of paramount importance because of their immutable nature. A Smart Contract is a program stored on a blockchain that runs when some predetermined conditions are met. While smart contracts have enabled a variety of applications on the blockchain, they may pose a significant security risk. Once a smart contract is deployed to a blockchain, it cannot be changed. It is therefore imperative that all bugs and errors are pruned out before deployment. With the increase in studies on Smart Contract vulnerability detection tools and methods, it is important to systematically review the state-of-the-art tools and methods. This, to classify the existing solutions, as well as identify gaps and challenges for future research. In this Systematic Literature Review (SLR), a total of 125 papers on Smart Contract vulnerability analysis and detection methods and tools were retrieved. These were then filtered based on predefined inclusion and exclusion criteria. Snowballing was then applied. A total of 40 relevant papers were selected and analyzed. The vulnerability detection tools and methods were classified into six categories: Symbolic execution, Syntax analysis, Abstract interpretation, Data flow analysis, Fuzzing test, and Machine learning. This SLR provides a broader scope than just Ethereum. Thus, the cross-chain applicability of the tools and methods were also evaluated. Cross-chain vulnerability detection is in this SLR defined as a method for detecting vulnerabilities in Smart Contract code that can be applied for multiple blockchains. The results of this study show that there are many highly accurate tools and methods available for Smart Contract (SC) vulnerability detection. Especially Machine Learning has in recent years drawn much attention from the research community. However, little effort has been invested in Smart Contract vulnerability detection on other chains.

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

Nostrud est velit minim laborum amet nisi enim est aute cupidatat eu amet Lorem. Enim elit ipsum culpa cupidatat officia. Aliqua minim veniam consectetur anim occaecat nulla commodo aute aliqua. Esse excepteur ullamco adipisicing quis nisi. Enim reprehenderit ex quis velit qui nostrud. Magna commodo elit labore sunt deserunt nostrud et mollit consequat nulla. Pariatur irure ut elit qui sunt veniam enim culpa consectetur id est commodo eu.

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

André Storhaug, Trondheim 28.05.2022

Contents

Figures

Tables

Code Listings

Acronyms

AST Abstract Syntax Tree. 18

BERT Bidirectional Encoder Representations from Transformers. 4

BLEU score BiLingual Evaluation Understudy score. xvi, 18, 19, *Glossary:* BiLingual Evaluation Understudy score

BLEU BiLingual Evaluation Understudy. xvi, 9, 10, 18–20, *Glossary:* BiLingual Evaluation Understudy

BPE Byte-Pair Encoding. 39

GPT General Pre-trained Transformer. 4

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

LSTM Long Short-Term Memory. 18

ML Machine Learning. vii, 1, 3, 14

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

PCFG Probabilistic context-free grammar. 17

RNN Recurrent Neural Network. 4

RoPE Rotary Position Embedding. 37, 39

SC Smart Contract. vii, xiii, xv, 2, 3, 11–14, 25–27, 30, 33, 35, 36, 49, 50

SLR Systematic Literature Review. vii, 50

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

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

Glossary

BiLingual Evaluation Understudy Metric for automatically evaluating machine-translated text. 9, 10, 18–20

BiLingual Evaluation Understudy score Metric for automatically evaluating machine-translated text. 18, 19

docstring Python function documentation strings. 18

F1 Harmonic mean of precision and recall. 18

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

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. Copilot uses these models to generate code for a given programming language. The tool is based on a deep learning model, named Codex 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, a code generation tool for generating novel code solutions to programming competitions.

A machine learning model is only as good as the data it is fed. These large-scale transformers needs huge amounts of data in order 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 extreme caution.

To better secure automatic code generation, this thesis 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 (Solidity). Smart contracts 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 [1]. Further, SCs tends to be rather short and simple, making it a good fit for generated code. The research questions addressed in this thesis are:

- RQ1 How to generate secure code with transformer-based language models?
- RQ2 How to generate smart contract code?

The specific contributions of this thesis are as follows:

- The currently largest smart contract dataset.
- 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 8 discuss the findings. Identified future work is presented in Chapter 9. Chapter 10 presents final remarks and concludes the thesis.

Chapter 2

Background

This chapter introduces the necessary background information for this study. First, a brief introduction to blockchain technology is provided in Section 2.9 and then the concept of Smart Contracts (SCs) is introduced in Section 2.10. Finally, in Section 2.10.1, the most popular SC vulnerabilities are described.

2.1 Natural language processing

2.1.1 Text preprocessing

2.1.2 Vector space model

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.

Term frequency-inverse document frequency

2.2 Machine Learning

Rewrite

Machine Learning (ML) is the study of computer algorithms that can automatically improve through the use of data. ML algorithms create a model based on training data. This model is then used to predict the outcome of new unseen data without being explicitly programmed to do so.

2.2.1 Machine learning models

2.2.2 Linear models

Logistic regression

Support vector machines

Artificial neural networks

2.2.3 Deep learning

Finish with RNNs and LSTM.??

2.3 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.* [2]. Unlike more traditional attention-based models such as 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.

2.3.1 Attention

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...*

2.3.2 Architecture

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

Figure 2.1: TODO

former model.

Tokenization

The Transformer takes in a sequence of tokens. These are generated input to the transformer is text is first parsed into tokens by a byte pair encoding tokenizer

Embedding and Positional Encoding

For the Transformer to process the text input, the text is first parsed into tokens by a byte pair encoding tokenizer. The model then needs 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.

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.

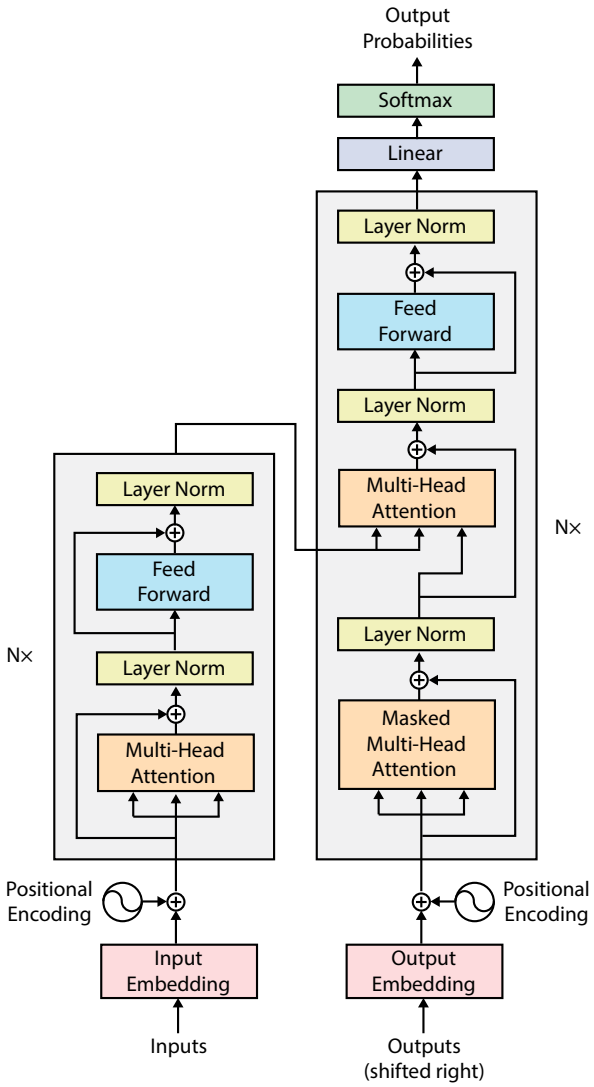


Figure 2.2: Architecture of a standard Transformer Vaswani *et al.* [2]

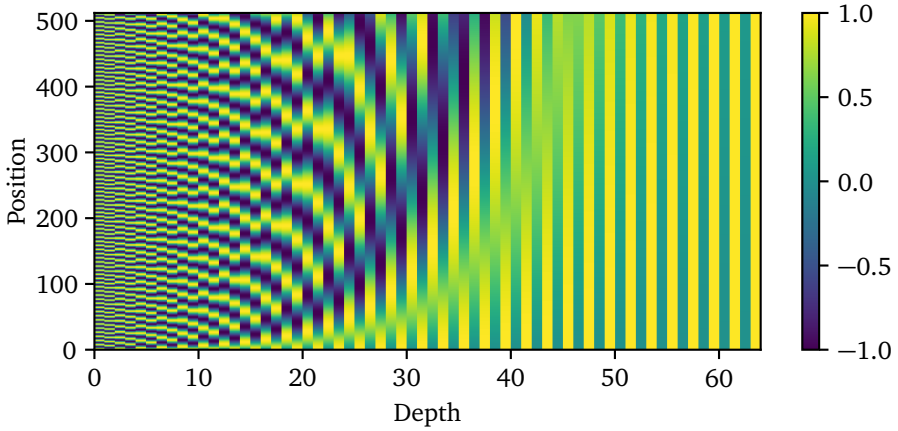


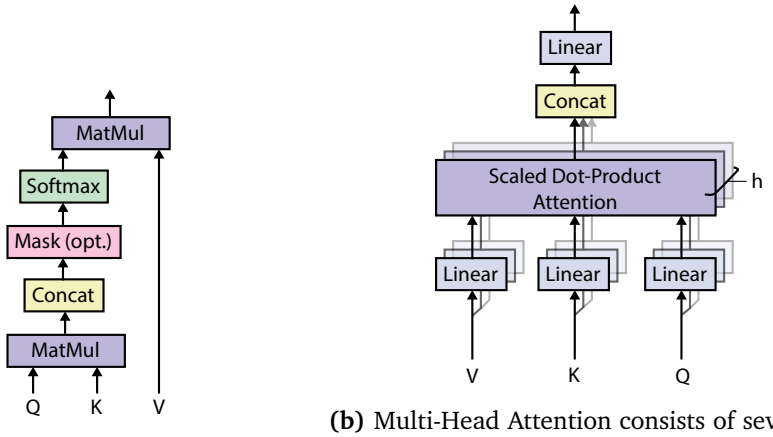
Figure 2.3: The 64-dimensional positonal encoding for a sentence with the maximum lenght of 512. Each row represents the embedding vector p_t

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 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.* [2] 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.



(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.* [2]

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 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:

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

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. 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.

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

2.3.4 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 ouput is available at the firs timestep, the decoder is fed a special "<start>" token. The decoder output is then fed back into the decoder again. This process is repeated until the decoder output encounters a special "<stop>" token.

2.4 Data parallelism

2.5 Performance metric

2.5.1 Precision

2.5.2 Recall

2.5.3 F1-score

2.6 Machine translation metrics

2.6.1 BLEU

BLEU (BiLingual Evaluation Understudy) by Papineni *et al.* [3] 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:

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

2.6.2 CodeBleu

https://arxiv.org/pdf/2009.10297.pdf
Maybe implement this for Solidity??

2.6.3 ROUGE

2.7 String metric

2.7.1 Jaccard index

2.8 Cluster Analysis

2.9 Blockchain

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 presents a distributed computing system with high Byzantine fault tolerance [4].

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. 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. If the platform also supplies a native currency (cryptocurrency), this is also shown.

2.9.1 Ethereum blockchain

2.10 Smart Contract

The term "Smart Contract" was introduced with the Ethereum platform in 2014. A Smart Contract (SC) is a program that is executed on a blockchain, 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,

Rewrite and adapt to vulnerabilities SoliDetector can detect.

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." [5]. The gas is used to pay for the cost of running the smart contract. This protects against malicious actors spamming the network [5]. 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.

2.10.1 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:

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.

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.

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

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

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.

Code listing 2.1: Access control vulnerable Solidity Smart Contract code

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

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 }
```

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 }
```

2.11 Vulnerability detection

Remove section

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.

2.11.1 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.

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.

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.

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.

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.

Make a literature review based on code synthesis / code auto completion. This would include both the main topic "code synthesis", as well as code comments for optimal code completion.

Code completion vs code synthesis vs automatic code generation vs intellisense.

Python: docstring - structured comment first line under function definition C++: Doxygen Java: javadoc JS: JSDoc

TS=("code" AND ("completion" OR "generation" OR "synthesis"))

TS=("code" AND ("auto-completion" OR "generation" OR "synthesis") AND comment)

spsp

3.1 Language models

The problem of generating code is fundamentally a language modeling problem. Language modeling is the task of predicting the next word in a text given the previous words. This section begins with presenting some of the earlier techniques, followed by surveying more recent and state-of-the-art language models.

The first few language models came in the form of n-grams, a term first referenced by Shannon [6]. An n-gram is a contiguous sequence of n items from a given sample of text. Most early approaches employed n-grams with smoothing to handle unseen n-grams Kneser and Ney [7].

3.1.1 Non-context models

3.1.2 Context aware models

3.1.3 Word embeddings

Tomas Mikolov's Word2vec (google team), Stanford University's GloVe GN-GloVe (genderneutral) ->point tto thte bias problem off datasets -> link to insecure code on github

3.1.4 Neural language models

N-grams

CoVe (Contextualized Word Vectors) needs "fixed" prettrained dataset :: Learned in Translation: Contextualized Word Vectors

ELMo biLM

Universal Language Model Fine-tuning (ULMFiT) :: introduced the concept of fine-tuning the language model.

OpenAI's Gener- ative Pre-training Transformer (GPT)

GPT-2, improved version of GPT. Works withoutfinettuning. (concerns of it being used to generate unintended or malicious content - delayed release)

Bidirectional Encoder Representations from Transformers (BERT) - > bidirectttional, in comparison to GPT

BERT is a bimodal Transformer with 12 layers, 768 dimentional hidden states, and 12 attention heads.

—— GPT-3 (Codex)

GPT-J (Opensoursed)

3.2 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) [8].

Hindle *et al.* [9] 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.* [10] 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.

3.2.1 Code semantics

Programs can also be synthesized by leveraging the semantics of the code. Alon *et al.* [11] 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 [12] builds on the works of Alon *et al.* [11] 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.* [13] 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%.

3.2.2 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.* [14] 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 [14], 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 [15] 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 finetuned 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 [3] of 28.69 for the model trained from scratch, and 32.79 for the finetuned version. Another model version based on GPT-2 is GPT-C by Svyatkovskiy *et al.* [16]. 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 BiLingual Evaluation Understudy score (BLEU score) precision of 0.80 and recall of 0.86. PyMT5 Clement *et al.* [17] 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

citation

BLEU score F-score of 24.8 on the CodeSearchNet test set.

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 pretrained version of GPT-J is also publicly available. Codex by Chen *et al.* [18] 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 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 averaged 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.

3.3 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??

3.4 Smart contract code generation

Some old-school models? I know various methods for programming smart contracs exists.. Maybe some graphical solutions?

3.5 Datasets

CodeXGLUE - Microsoft

CodeNet - IBM

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

Table 3.2: Existing code datasets.

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.

autoregressive generation tasks see https://arxiv.org/pdf/2005.08025.pdf for structuring thesisl....!!!!

see docstring analysis 2.3 of https://arxiv.org/pdf/2010.03150.pdf for clustering comments....

3.6 Code comment analysis

Several papers on generating code comments from source code are available. The following is a list of the most popular papers.

However, to the best of my knowledge, there is no paper investigating how to best write comments for auto-generating code. There are however,

3.7 Vulnerability detection tools

Include vulnerability detection tools table from literature reeview

SmartBugs

Most of the available tools only work on special versions of Solidity. ... Why use SoliDetector?

Chapter 4

Research Methodology

This chapter presents the research method used in this thesis. Firstly, the research motivation is presented, followed by the research questions defined for this thesis. The research method and design are explained in the third section. The fourth section presents an overview of the research implementation. Finally, the fifth section presents the various tools and libraries used for 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]. Recent works have applied transformers for code generation and program synthesis, achieving state-of-the-art results. For example, Codex by [18] 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 [18], 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" [21]. An empirical study by Pearce *et al.* [22] 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 tries to address the above problems by answering the research questions defined in Section 4.2. this includes both investigating how

find real number here .

Rewrite

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?

RQ1.1. How well do pre-trained transformer-based language models work for smart contract synthesis?

RQ1.2. What impact does fine-tuning a transformer model on smart contract code have on its code generation ability?

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

RQ3. How to best construct inputs for automatic code generation?

compare to literature review in chapter: related-works

4.3 Research Method and Design

The underlying foundation for how research is conducted is rooted in the research philosophy used. For this research, a positivistic research philosophy was used. A positivistic philosophy assumes that the world is not random, that it is ordered and regular, and that one can investigate it objectively. A deductive research approach was used. To best facilitate the answering of the research questions defined in Section 4.2, experiments were selected as the research strategy. These experiments included fine-tuning pre-trained language models, as well as testing these on real data and man-made data. The results from the evaluation of the fine-tuned models are recorded as observations and quantitatively evaluated.

Cite - B. J. Oates, Researching information systems and computing, Sage, 2006.

4.4 Research Implementation

For the implementation of the research, first, the datasets needed for the experiments were constructed. The datasets creation phase is divided into the following steps:

1. Create verified smart contract source code dataset.
    a. Scrape verified smart contracts from the Ethereum blockchain.
    b. Filter scraped verified smart contracts for uniqueness.
2. Create an audited version of the smart contract dataset
    a. Label the smart contracts with a vulnerability detection tool.
3. Create a parsed dataset containing "comment, function" pairs. This will facilitate testing, as well as research question 3.
    a. Create a parser that can parse all contract versions.
    b. Parsing the verified smart contracts with a parser.

A comprehensive description of the creation of the datasets used in this project is given in Chapter 5.

Secondly, the language modeling process is divided into 2 phases:

1. Fine-tune a transformer model on the verified smart contracts dataset.
2. Fine-tune a transformer model on the audited verified smart contract dataset, employing security conditioning.

A comprehensive description of the implementation of the language models is given in Chapter 6.

4.5 Plan of the Experiments

This section presents the plan for how the experiments in Chapter 7 are executed. This includes an overview of the different technologies applied, both software and hardware, as well as the different phases of the experiments.

4.5.1 Technology

4.5.2 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 [**transformers**] library by Hugging Face [**hugging-face**] fulfilled these conditions. The library provides flexible and easy-to-use solutions. It also supports integration with DeepSpeed [**deepspeed**], a deep learning optimization library by Microsoft [**microsoft**] 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.

4.5.3 Hardware resources

IDUN High Performance Computing Platform

4.5.4 Experimental process

This project employs an experimental and analytical research method, as described in Section 4.3. The standard scientific method was used for all the experiments. This included stating a hypothesis, implementing the required systems, executing the experiments, and evaluating the results. Several metrics were used to evaluate the results. For evaluating the performance of the smart contract generation, accuracy, BLEU score and Perplexity were used.

in addition to the standard deviation when conducting statistical hypothesis tests.

Figure 4.1: Image of IDUN todo: add ref `https://www.hpc.ntnu.no/idun/`

These experiments included fine-tuning pre-trained language models, as well as testing these on real data and man-made data

4.5.5 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 6.2), no hyper-parameter optimization was performed. The hyper-parameters were left to defaults used during pre-training. Hence, everything but the training data is keept 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.

Chapter 5

Data

This chapter introduces the necessary background information for this study. First, a brief introduction to blockchain technology is provided in Section 2.9 and then the concept of Smart Contracts (SCs) is introduced in Section 2.10. Finally, in Section 2.10.1, the most popular SC vulnerabilities are described.

5.1 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 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 reestarrting, 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

5.1.1 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.

5.1.2 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

Reference libraries

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.

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.

5.2 Datasets

This section describes the datasets used and created in this study.

5.2.1 The Pile

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

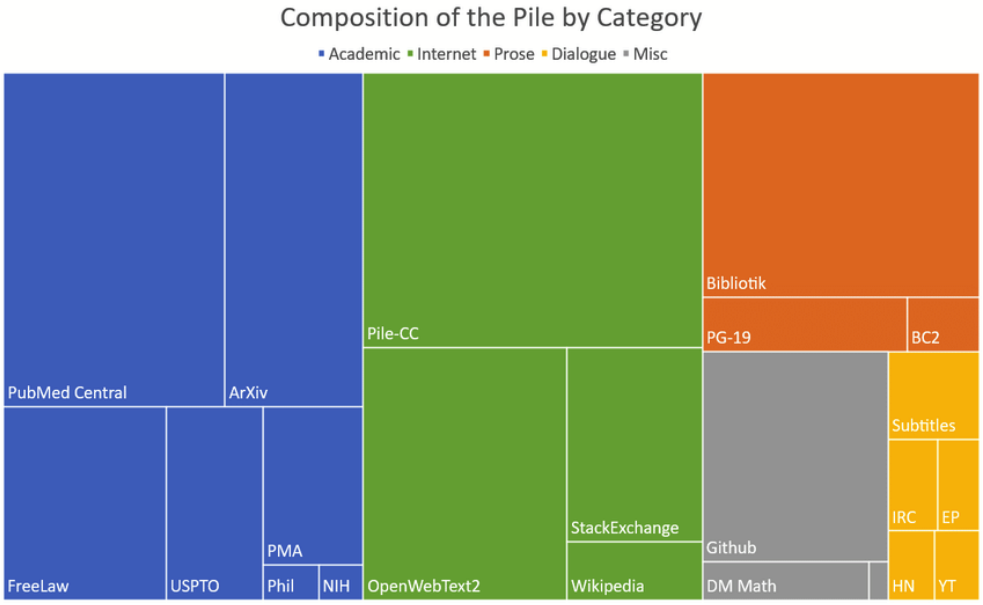


Figure 5.1: Treemap of Pile components by effective size. SOURCE FROM THEPILE paper

5.2.2 Verified Smart Contracts

`https://github.com/andstor/verified-smart-contracts` `https://huggingface.co/datasets/andstor/smart_contracts`

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

The Verified Smart Contracts dataset is a dataset consisting of verified Smart Contracts from Etherscan.io. This is real smmart contracts that are deployed to the Ethereum blockchain. A set of 100,000 to 200,000 contracts are provided, containing both Solidity and Vyper code.

Table 5.1 shows the metrics of the various (sub)datasets.

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. 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 5.2.2. 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 libray 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.

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.

```
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',
```

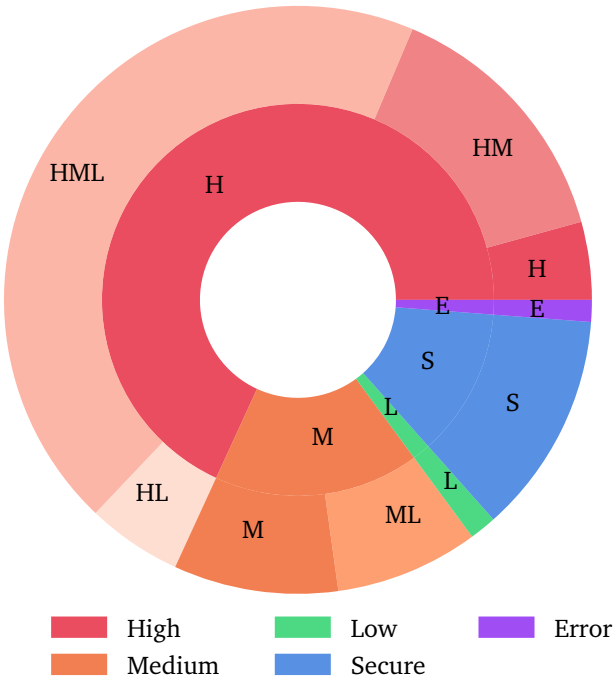


Figure 5.2: Doughnut chart over the distribution of the vulnerability severities in the flattened dataset at different granularity levels, where each level occurs at least once in the SC. The outer ring shows the additional security levels for each contract. For example, "HML" means that the contract has at least three vulnerabilities with the corresponding "High", "Medium", and "Low" security levels.

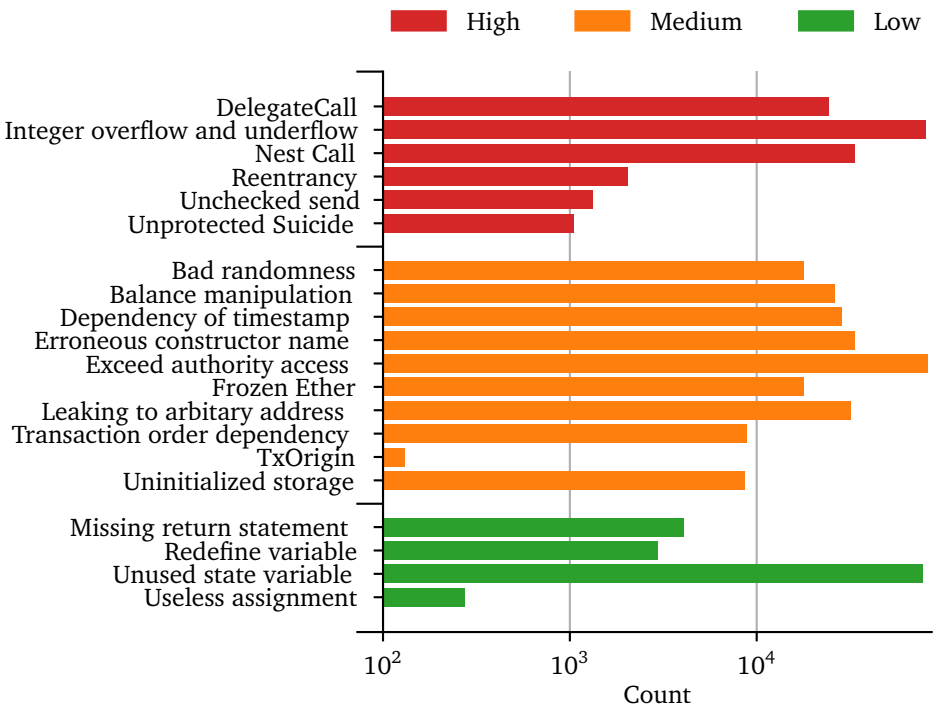


Figure 5.3: Distribution of vulnerabilities in the flattened dataset.

```
    'file_path': 'PinkLemonade.sol',
    'contract_address': '0x9a5be3cc368f01a0566a613aad7183783cff7eec',
    'language': 'Solidity',
    'source_code': '/**\r\n\r\nt.me/pinklemonadecoin\r\n*/\r\n\r\n// SPDX-
↪ License-Identifier: MIT\r\npragma solidity ^0.8.0;\r\n\r\n\r\n/*\r\n * @dev
↪  Provides information about the current execution context, including the\r\
↪ n * sender of the transaction and its data. While these are generally
↪ available...',
    'abi': '[{"inputs":[],"stateMutability":"nonpayable","type":"constructor"}
↪ ...]',
    'compiler_version': 'v0.8.4+commit.c7e474f2',
    'optimization_used': False,
    'runs': 200,
    'constructor_arguments': '',
    'evm_version': 'Default',
    'library': '',
    'license_type': 'MIT',
    'proxy': False,
    'implementation': '',
    'swarm_source': 'ipfs://eb0ac9491a04e7a196280fd27ce355a85d79b34c7b0a83ab606
↪ d27972a06050c'
  }
```

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.

Parsed

5.2.3 Verified Smart Contracts Audit

`https://github.com/andstor/verified-smart-contracts-audit` `https://huggingface.co/datasets/andstor/smart_contracts_audit`

Subsets:

SoliDetector

5.2.4 Smart Contract Comments

`https://huggingface.co/datasets/andstor/smart_contract_comments` See **??** for more information.

Figure 5.4: 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. The inner ring shows the distribution of the occurrences of each level. 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.

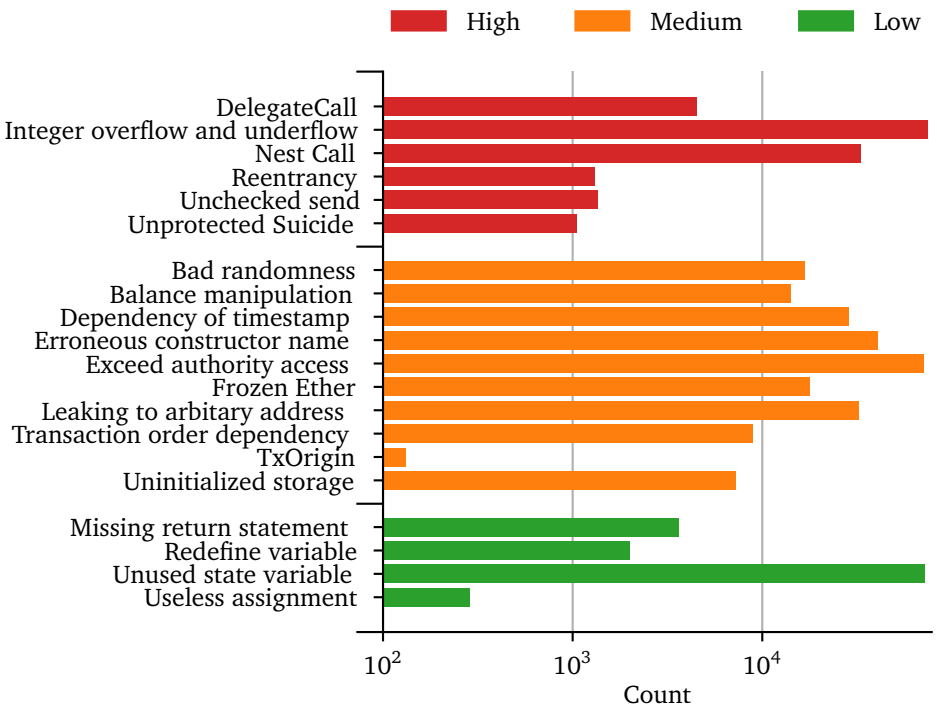


Figure 5.5: Distribution of vulnerabilities in the inflated dataset.

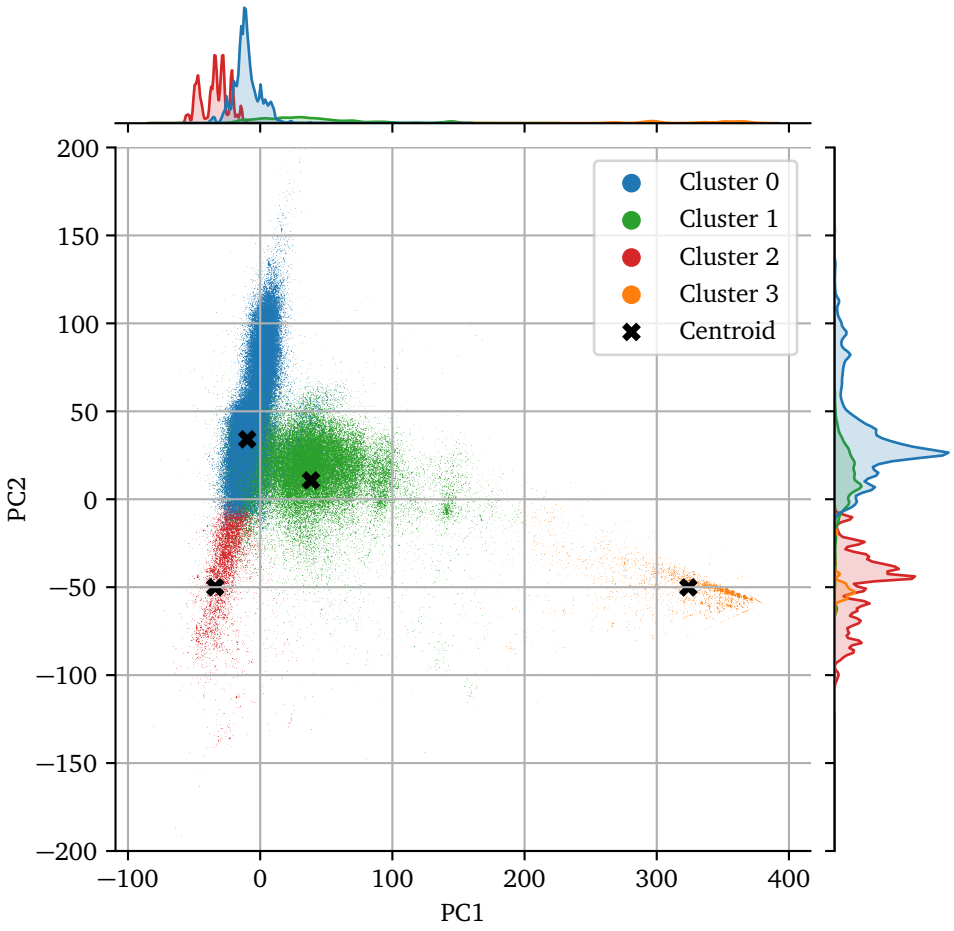


Figure 5.6: Histogram x of verified SCs on Ethereum

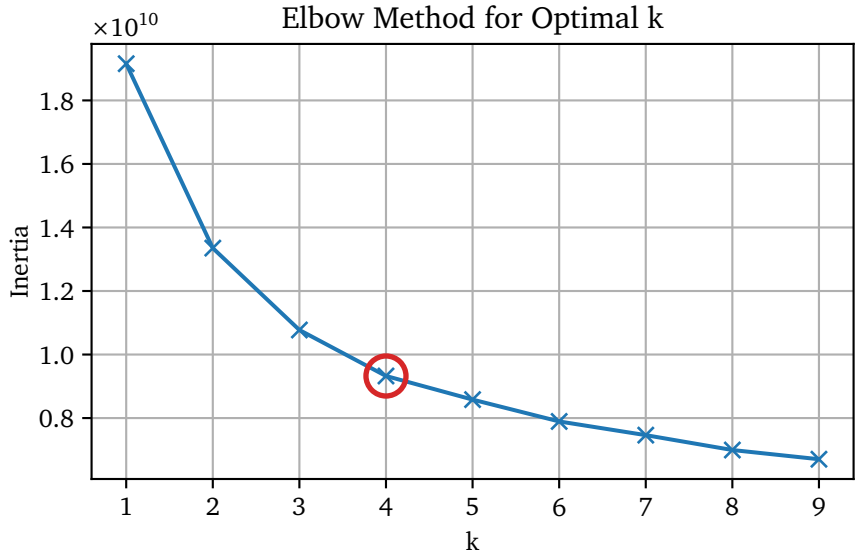


Figure 5.7: Histogram x of verified SCs on Ethereum

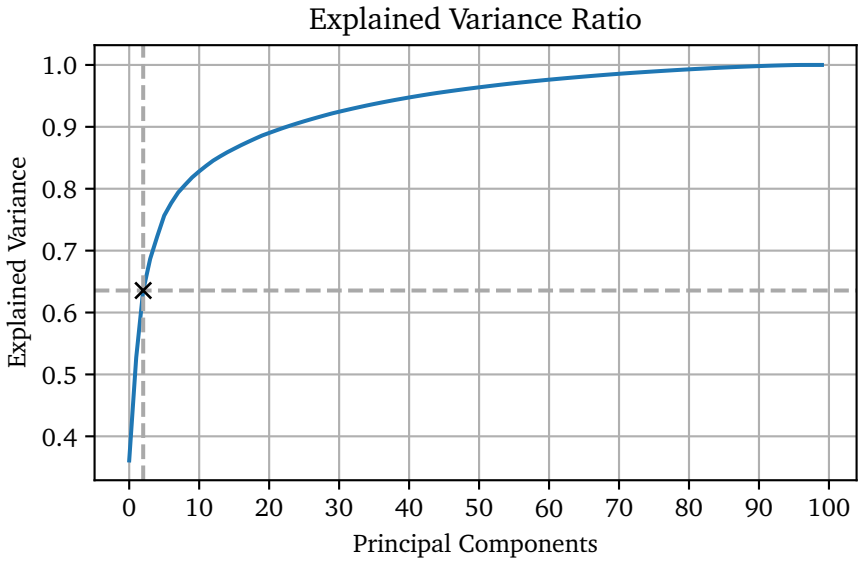


Figure 5.8: Histogram x of verified SCs on Ethereum

Chapter 6

Language Modeling

This chapter presents the results of this thesis. The chapter starts with ... the research questions defined in Section 4.2.

As discussed in section Section 3.2.2, there are several available transformer models. However, only a few of them have open-sourced pre-trained weights. Of these, only GPT-J [23] includes code in it's pre-training dataset "The Pile", described in Section 5.2.1. As GPT-J is a state-of-the-art generative pre-trained transformer model, this is the language model used in this thesis. This chapter presents a detailed overview of the system architecture for generating secure Smart Contract code. The first section gives an overview of the GPT-J model architecture, followed by a section describing the pre-training of the model. The third section describes the fine-tuning process on the smart contract dataset presented in Sections 5.2.2 and 5.2.3.

Add figure of training process

6.1 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 [23] 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 [24], 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 5.2.1 for a more detailed description of the Pile.

Being a GPT class transformer, GPT-J uses a decoder-only architecture, as can be seen in Figure 6.1. 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)

[25] for position encoding. Opposite to sinusoidal encoding used in standard transformer models (see Section 2.3.2), this is shown to result in better model quality in tasks with long text [25].

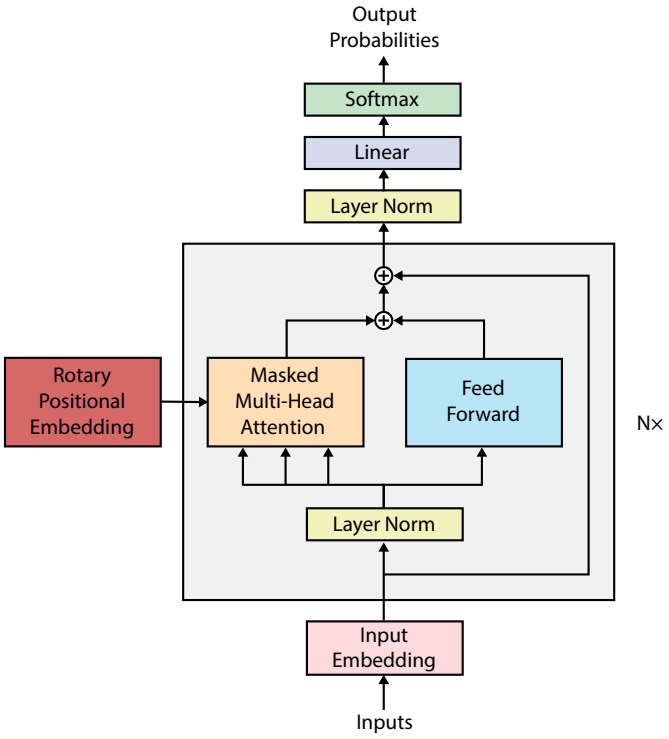


Figure 6.1: Diagram of GPT-J model architecture

6.2 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)

Table 6.1: GPT-J-6B model details.

Hyper parameter	**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

is available, one can use mixed precision training with PyTorch 1.6.0 or later, or by installing the Apex library for previous versions. Just add the flag –fp16 to your command! If you have an NVIDIA “Ampere” GPU architecture, you can use Brain Floting Point (BF16) by passing the flag –bf16. Using mixed precision training usually results in 2x-speedup for training with the same final results.

6.3 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. In this project, pre-trained weights for GPT-J-6B from ElutherAI are used. The pre-training by ElutherAI is done on the dataset The Pile, described in Section 5.2.1. Of the roughly 825GiB, 95.16 GiB (7.59%) of The Pile is code from GitHub. Compared to many other open-source models, GPT-J-6B is one of the most promising models for the task of code generation.

The specific GPT-J model configuration can be seen in Table 6.1. 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.

* each layer consists of one feedforward block and one self attention block

add table notes

6.4 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, two models are created. The first model, named GPT-J-6B-Smart-Contract, is fine-tuned on the Verified Smart Contracts dataset Section 5.2.2. The other model, named GPT-J-6B-Smart-Contract-Audit, is a secure version of the first model. It is fine-tuned on the Verified Smart Contracts Audit dataset Section 5.2.3, the same dataset as for the first model but with additional labeling from vulnerability analysis.

6.5 Inference

TODO: What is supported by the model? How much memory to use? We perform beam search with width of 5 and optimize for accuracy@1

6.6 Security Conditioning

When training a large language model on several gigabytes of open-source code, it is safe to assume that large portions of this code are not safe and contains vulnerabilities. In the case of Smart Contracts, the vulnerability analysis presented in section 5.2.2 shows that almost 50% of deployed Smart Contracts contain 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.

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 5.2.3.

find correct wording.

Cite previous works

Add example of security conditioning.

Table 6.3: Hyper parameters for GPT-J model

Hyper parameter	
_name_or_path	EleutherAI/gpt-j-6B
activation_function	gelu_new
architectures	GPTJForCausalLM
attn_pdrop	0.0
bos_token_id	50256
embd_pdrop	0.0
eos_token_id	50256
gradient_checkpointing	false
initializer_range	0.02
layer_norm_epsilon	1e-05
model_type	gptj
n_embd	4096
n_head	16
n_inner	null
n_layer	28
n_positions	2048
resid_pdrop	0.0
rotary	true
rotary_dim	64
scale_attn_weights	true
summary_activation	null
summary_first_dropout	0.1
summary_proj_to_labels	true
summary_type	cls_index
summary_use_proj	true
tie_word_embeddings	false
tokenizer_class	"GPT2Tokenizer"
transformers_version	"4.19.0.dev0"
use_cache	true
vocab_size	50400

Table 6.5: DeepSpeed Zero config.

Hyper parameter	
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

Chapter 7

Experiments and Results

This chapter presents the experiments and results that have been conducted. The experiments are grouped by research question. First, experiment **E1** takes on research question 1 regarding automatic smart contract code synthesis. It compares the performance of pre-trained transformer-based language models on the verified smart contract dataset. Experiment **E2** focuses on the security of smart contract code generation, tackling research question 2. It compares the security of a fine-tuned model utilizing security conditioning purposed in Section 6.6. This section presents three sub-experiments:

- **E2.1.** Compares security performance based on providing entire class context from audited verified smart contract dataset.
- **E2.2.** Compares security performance based on using only comments as input context from audited verified smart contract dataset.
- **E2.3.** Compares security performance based on testing a custom vulnerability-inclined dataset.

The last experiment **E3** takes on research question 3 regarding how to best formulate the input for automatic smart contract code synthesis.

7.1 E1 - Automatic Smart Contract Code Synthesis

Goal

Method and Data

Results and Discussion

7.2 E2 - Security Conditioning

This section presents the results and discussion of the sub-experiments of experiment **E2**. Experiment **E2** focuses on the security of smart contract code generation, tackling research question 2. It compares the security of a fine-tuned model with

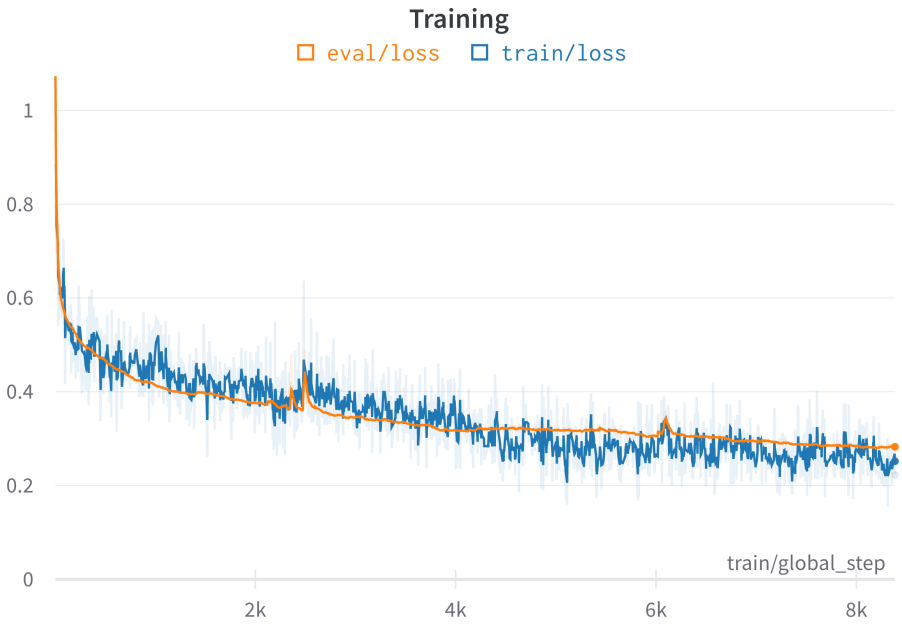


Figure 7.1: Training and evaluation loss during model training.

Figure 7.2: Evaluation accuracy during model training.

and without utilizing security conditioning purposed in Section 6.6. The following three sub-experiments are all based on the same fine-tuned models, but uses different evaluation method.

Goal

Method and Data

Results and Discussion

7.2.1 E2.1 - Complete Class Context

Goal

Method and Data

Results and Discussion

7.2.2 E2.2 - Comment Context

Goal

Method and Data

Results and Discussion

7.2.3 E2.3 - Inclined Vulnerabilities

Goal

Method and Data

Results and Discussion

7.3 E3 - Formulation of Inputs

Goal

Method and Data

Results and Discussion

7.3.1 InclinedVulnerabilities

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

Move relevant stuff below to method chapter (experiment plan)

Move to datasets?)

7.4 Baselines

7.4.1 InclinedVulnerabilities

7.5 Evaluation metrics

Rewrite

Accuracy could measure correctness of the exact match, failing, however, to capture the proximity when a completion suggestion partially matches the target sequence, which could still be a valid completion suggestion.

Rewrite

The ROUGE score is the metric commonly used to evaluate machine translation models. Its ROUGE-L variant is based on the Longest Common Subsequence (LCS) statistics. LCS takes into account structure similarity and identifies longest co-occurring n-grams.

Rewrite

The Levenshtein distance measures how many single-character edits including insertion, substitution, or deletion - does it take to transform one sequence of tokens to another. Quite often, even if a suggested completion is only an approximate match, developers are willing to accept it, making appropriate edits afterwards. As such, the Levenshtein edit similarity is a critical evaluation metric.

Get logits from a model prediction to visualize the distribution of the predicted probabilities.

7.6 Quantitative evaluation

Even though only "HIH" severity vulnerabilities are labeled, several most of these also contain medium and low severity vulnerabilities. See Doughnut chart... The "full" context code is subject to latent vulnerabilities, which are unavoidable. Hence this could explain the low decrease in vulnerabilities.

7.7 Qualitative evaluation

7.8 Memorisation evaluation

Check if the model is just copying

Do this by investigating logits from all layers.

Check common substrings in dattaset.

7.9 Diff

Tryto make custom generattion function that only selects SECURE solutions.....??

Does the temperature affect how secure solutions are?

Huugging face emojie as transformer model!

7.10 Model weigts

THIS USES AN INDUCTIVE ANALYSIS. Fix methodology chapter. ADD as Experiment 4?

Can we find structures in the eriht? Neural view? bertviz? That resembles AST equivalent?

Answers to research questions Evaluation of the answers

Chapter 8

Discussion

In this thesis, ...

8.1 Comparison with related work

Compared to related ...

8.2 Threats to Validity

Data contamination in test and train datasets.

Chapter 9

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.

9.1 Comparison with related work

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

9.2 Threats to validity

Faults in SoliDettector...

Future work, combine multiple vulnerability detectors

Chapter 10

Conclusion

This paper presents the results of a Systematic Literature Review of existing Smart Contract vulnerability analysis and detection methods. The motivation for this research was to provide a state-of-the-art overview of the current situation of the SC vulnerability detection. A total of 40 primary studies were selected based on predefined inclusion and exclusion criteria. A systematic analysis and synthesis of the data were extracted from the papers, and comprehensive reviews were performed. Further, to the greatest extent, this paper also identifies the current available cross-chain tools and methods. The cross-chain applicability for these assets is investigated and analyzed.

The findings from this study show that there are a number of methods and implemented tools readily available for vulnerability analysis and detection. Several of these tools show great results. The most prevalent methods are static analysis tools, where symbolic execution is among the most popular. Other methods such as syntax analysis, abstract interpretation, data flow analysis, fuzzy testing, and machine learning are also readily used. In this paper, some potential cross-chain tools are highlighted and discussed. Although they pose several limitations, they show significant potential for further development. Especially interesting is the potential machine learning-based cross-chain detection methods.

From this study, one can see that there is a significant lack of research on vulnerability detection on other blockchain platforms than Ethereum. The hope is that the results from this study provide a starting point for future research on cross-chain analysis.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[7] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling," in *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, 1995, 181–184 vol.1. DOI: 10.1109/ICASSP.1995.479394.

[8] 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/.

[9] 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.

[10] 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/.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[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] GitHub. "Your ai pair programmer." (2022), [Online]. Available: https://github.com/features/copilot (visited on 07/01/2022).

[22] 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.

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

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

[25] 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.

André Storhaug

Secure Smart Contract Code Synthesis

NTNU
Norwegian University of
Science and Technology