POLITECNICO DI TORINO

ICT for Smart Societies

Master's Degree Thesis

Smart Contracts and Solidity Code Summarization

supervisors:
Ing. GATTESCHI VALENTINA
Prof. LAMBERTI FABRIZIO

candidate:
ID S267461
MATTEO ZHANG

A.Y.2020-2021

Contents

Summary

Blockchain became a hot topic in the last decade. Blockchain is the underlying technology enabling Bitcoin transfers. In particular, this technology ensures the integrity of digital records and enables the transfer of decentralized digital currency. After the creation of bitcoin, Vitalik Buterin saw the unexpressed potential of this technology in other fields that go beyond a simple transfer of value and created Ethereum, an open-source, decentralized blockchain with smart contract functionality. Consequently, smart contracts became the centre of the Blockchain economy.

Smart contracts are programs that run on the blockchain. In the Ethereum case, the most used programming language to code smart contracts is Solidity, a relatively new programming language. One of the main problems in computer science is source code documentation, and this is remarkably true for Solidity. When Solidity developers need to understand smart contracts code, the code generally lacks comments and proper documentation. In order to provide code documentation to programmers and provide a coherent summary of the source code in the Blockchain context, this thesis presents a system to automatically generate comments for Solidity smart contracts code. The created system "SolidparticySummarizer" will significantly help beginners understand the code, given that the developers spend most of their time on this task. Furthermore, the system could also be useful for people with limited coding skills, according to our final survey, since it will help them understand the source code of smart contracts with detailed summaries.

In the first part of the thesis, the state of the art in solidity code summarization is analyzed. In the second part, the state of the art in general code summarization techniques are presented. Then, the developed system is described. Such system relies on transformer models to perform natural language generation from source code. The thesis also presents the dataset created to train and test the system, which was created by collecting and cleaning Solidity smart contracts. Finally, results show that the created tool provides better results than the current state of the art solidity document generation tools.

Acknowledgements

I would like to thank my supervisors, Professor Fabrizio Lamberti and Valentina Gatteschi, for guiding me throughout the thesis research questions and methodology. I also want to thank Taha Zafar who advised me on research topics and friends who partecipated in the final survey. Finally I want to thank my family and Huifeng Tan who supported me in my master years.

Chapter 1

Introduction

Smart contracts are programmed self-executing contracts that run on a blockchain without the supervision of third parties like lawyers. Smart contracts are possible thanks to the peer-to-peer computer network that hosts the distributed ledger called blockchain. There are different types of blockchains, but they all share a typical structure: blocks of information linked together by cryptography. The majority of smart contracts are deployed on the Ethereum blockchain, which is one of the biggest blockchains around the world. Smart contracts are programmed in Solidity, which is a new programming language proposed in 2014 and released in 2015 [1]. From one end research on blockchain technology is very active and on the other end active attempts of research on readability of smart contracts for non-experts are ongoing. This thesis is inserted inside this context since the scope is to automatically generate documentation for the source code. In particular, the objective of the thesis is to create a system - the SolidparticularSummarizer - that is able to automatically generate smart contracts code documentation.

The thesis is structured as follows:

In the first chapter, the blockchain is introduced. The technology became popular starting from the Bitcoin white paper[2]. Then a new major blockchain, the Ethereum blockchain, is created to exploit smart contracts potential, unexpressed by the Bitcoin blockchain. At the end of the chapter, specific problems in smart contract programming are presented and explained, and a portion of these problems can be solved with SoliditySummarizer.

In the second chapter, tools and techniques belonging to the state of the art are presented. SMTranslator represents the current state of the art in solidity code summarization. It is a tool programmed to extract information from smart contracts source code and combine it with templates to create code summaries. Other techniques are also available for the code summarization

task; in particular, this thesis explore state of the art machine learning techniques employing transformer models. These models are implemented later on in this thesis.

In the third chapter, SolidittySummarizer is implemented creating two code repositories: one is SmartContractDatabase and the other is a fork of the Transformer model implementation. To train a good Transformer model, the data are essential. In particular, this chapter will discuss the custom tool SmartContractDatabase, made for data gathering and cleaning as well as the Transformer model which is taken from an existing online repository.

The fourth chapter is about the evaluation of the SolidittySummarizer system. Especially, the generated comments are evaluated in both quantitative metrics and qualitative opinions of fellow programmers. The opinions are gathered through an online survey.

The fifth chapter presents the conclusions and highlights some future works.

1.1 Blockchain

Blockchain is the technology that enables the world most famous cryptocurrency, Bitcoin (Natamoto, 2008) [2], as the name suggests, it is a chain of information blocks linked together using cryptography, the information written in the blocks are immutable, making blockchain a perfect application for permanent records.

Digital records are used in every sector in the modern era, particularly the finance sector, that are in the majority centralized systems meaning only one central authority can change the ledger of accounts.

For this reason, the person behind the pseudonym "Satoshi Nakamoto" published the white-paper "Bitcoin: A Peer-to-Peer Electronic Cash System" [2], building upon the blockchain technology, he created a distributed ledger system where anyone can participate and join the peer to peer network, which guarantees the transaction settlement of Bitcoins and the ledger integrity without centralized institutions.

The integrity of a ledger in a blockchain is intrinsic to the block structure; the first block is the genesis block (block 0) fig. 1.1, following blocks are linked backward to it, after a block n is linked to a block n-1 and mined, new Bitcoins are created as a reward system for the miner. Mining a block means to validate and register the transactions of the last block to the blockchain.

The block structure fig. 1.1 is composed by the Header and Transactions, the header needs to record: the timestamp of the block creation in Unix epoch; the Proof of Work (POW) difficulty target (in the Bitcoin and Ethereum case); Nonce (number only used once) a random string to append to the current head hash; Merkle root hash.

The header also includes the parent block hash, which is the result of the SHA-256 algorithm applied on the previous block, the validation of the current block is done by mining which is solving the hash of the block such that it starts with a number of zero bits. This method of validation is called POW, and the time spent mining is proportional to the number of zeros. Compromising any block means to mine every subsequent block, which is surely slower than mining only the last block.

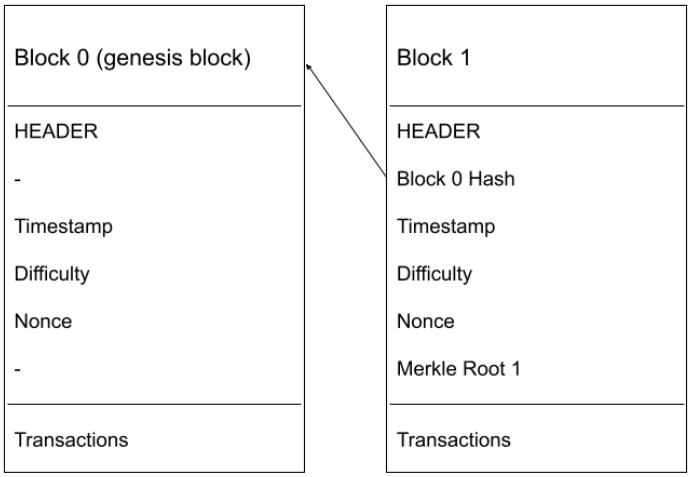


Figure 1.1: Block structure and Blockchain structure

The transactions inside a block are all collected inside the Merkle tree, and then they are hashed to obtain the Merkle Root inside a block; this way, it is not necessary to store the complete blockchain but only the heads in order to participate the peer to peer network.

The creation of a block is the process for a miner to listen to the network and collect the pending transactions. After collecting the transactions from a transaction pool, the miner proceeds to create a candidate block to insert in the blockchain, notice that other miners are doing the same, if the miner finds the POW of the block by mining it, then reward and fees are sent

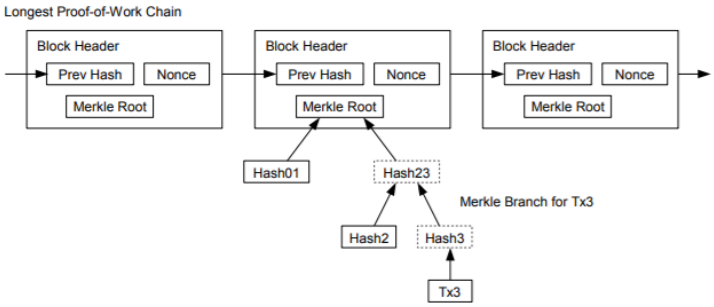


Figure 1.2: Merkle tree after pruning.
A person can check the transaction Tx3 by linking to the Merkle Root. [2]

to the miner's wallet. Each wallet address is identified with a public key, and every peer to peer transaction must be signed by the private key of the corresponding address.

1.2 Ethereum Ecosystem

The Bitcoin blockchain created a new era of electronic payments, where people can engage in transactions without a central authority. It solved many problems related to the topic, such as the "double spending" problem, electronic payment based on POW instead of a third-party institution or company, non-reversible transactions creating a permanent ledger always available to the public. However, along with the solutions, there are still many unexplored areas, such as smart contracts and decentralized web.

Vitalik Buterin in 2013 published the Ethereum white paper [1] to explore an alternative decentralized protocol for creating decentralized applications and smart contracts, which has limited application in the Bitcoin blockchain with Bitcoin Script.

"Bitcoin Script is essentially a list of instructions recorded with each transaction that describes how the next person wanting to spend the Bitcoins being transferred can gain access to them" [3]. On top of the Bitcoin blockchain, simple smart contracts can be built, but they are limited by the set of instructions called OPCODES.

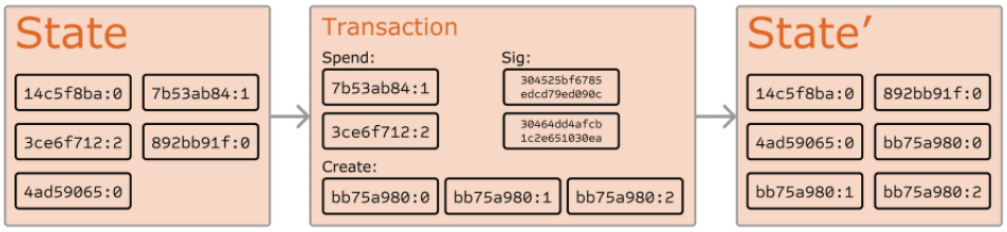
Bitcoin Script does not support loops, making this set of instructions not Turing complete; it does not provide hedging contracts which means transactions of bitcoin cannot be programmed when they are expressed in other currencies, e.g. send to Alice 100 US dollar worth in Bitcoin tomorrow; the transactions have no intermediate "states": they are either spent or

unspent, the Script cannot access to the block data such as Nonce and timestamp.

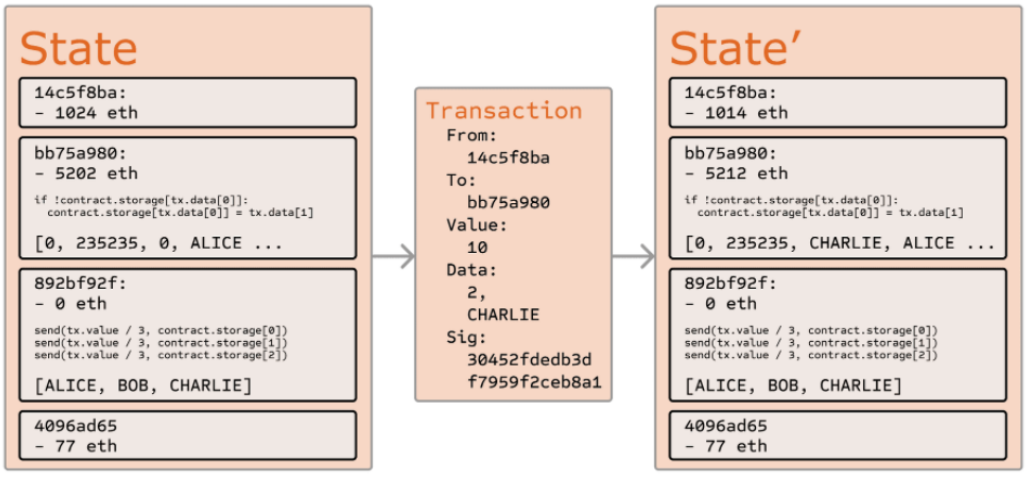
1.2.1 Ethereum Introduction

Ethereum is "a blockchain with a built-in Turing-complete programming language" [1], it is born with more computational power and more features than Bitcoin blockchain.

In Bitcoin a state S can be defined as ownership status exactly as a ledger, each state corresponds to wallet address and amount of Bitcoins, applying a transaction on state S the result is a new state S' with different balances or error message if any balance goes under zero fig. 1.3a.



(a) Bitcoin blockchain
described as transition states [1]



(b) Ethereum states
described as transition states [1]

Figure 1.3: States and State transitions [1]

In Ethereum States are called accounts fig. 1.3b, each account have an address

and accounts can be:

- externally owned - controlled by private keys.
- contracts - a smart contract deployed to the network, controlled by code.

An account contain nonce (differently from Bitcoin is intended to make transaction be processed only once), ether balance, contract code if present, account's storage empty by default.

An externally owned account has no code only ether balance, but from an externally owned account it is possible to send messages by creating and signing a transaction; in a contract account instead there is code and ether balance, every time the contract account receives a message, it activates its code, allowing itself to read and write from internal storage and to send messages or to create other contract accounts [1].

The contracts are called smart contracts because they can be seen as *"autonomous agents"* [1] while Bitcoin script contract are contracts that needs only to be fulfilled, e.g. like signing with multiple private key. The magnitude of operations are different.

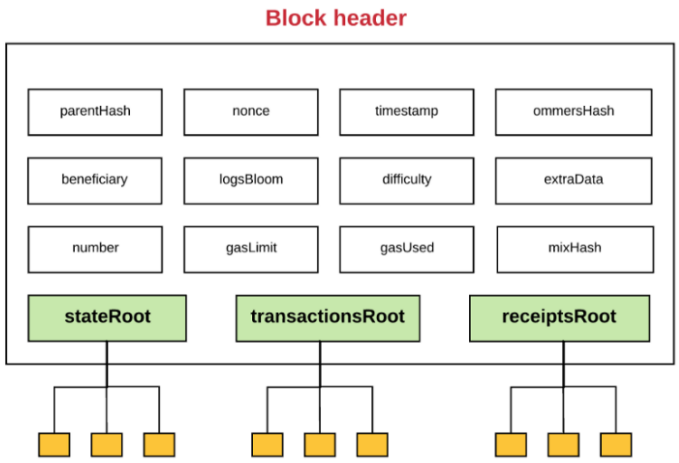


Figure 1.4: Ethereum Blockchain Block header and modified Merkle Patricia Tree roots.

The Ethereum network instead of a distributed ledger, it is a real state machine, the Ethereum State is a data structure called modified Merkle Patricia Tree present in every node of the network and linked together by

block heads fig. 1.1. The transaction tree is the same of the Bitcoin blockchain, the state tree represents the current states of the blockchain and the receipts tree represents the resulting state of each transaction.

The transactions are divided in two types, those which result in message calls and those which result in contract creation. Whenever an account sends a message or (internal transaction) to an already created and compiled contract, it executes its bytecode (compiled contract code). If the execution involve a state change in the blockchain then fees are required. The execution of the bytecode is done with the Ethereum Virtual Machine (EVM) which is implemented in every Ethereum client.

Gas is the measurement of computational complexity required to execute a specific operation with the EVM, in simple terms it is the fee required to conduct a transaction. The fee system actually is more complex because the Gas imply a gas price, the gas price is expressed in Gwei ($10^{-9}Eth$) and the total fee is equal to $Gas \times gasPrice$. The fee system discourage attackers to perform hostile attack loops and prevents accidental code bugs to loop forever, both loops require Eth and the amout of Eth inside contract account has a limit.

1.2.2 Smart Contracts

In this thesis the main focus is on Ethereum because its community is one of the biggest in the cryptocurrency world by market size, only second to Bitcoin. Solidity, Vyper, Yul/Yul+ and FE are cited in the Ethereum website [1] as main programming languages for Ethereum smart contracts. Solidity surpass by far the other 2 [4], empirically on Stack Overflow and Google trends Solidity is the most commonly used language, this fact is also confermed by Tiobe index [5] where Solidity is ranked around 90s top programming language.

In Ethereum a program that runs on the blockchain is called "smart contract", it is a contract account with its own address and balance, once deployed it can't be deleted by default and the transactions to and from the smart contract are irreversible. On a higher level smart contracts can be seen as a protocol to execute automatically the contractual terms, the execution takes place inside the EVM which is on top of the blockchain.

Building on top of the Ethereum blockchain means anyone can deploy a smart contract, it just need to be coded with one of the programming language like Solidity. Then the deployed contract is compiled into Ethereum bytecode in order to be executed. A contract can even deploy other contracts

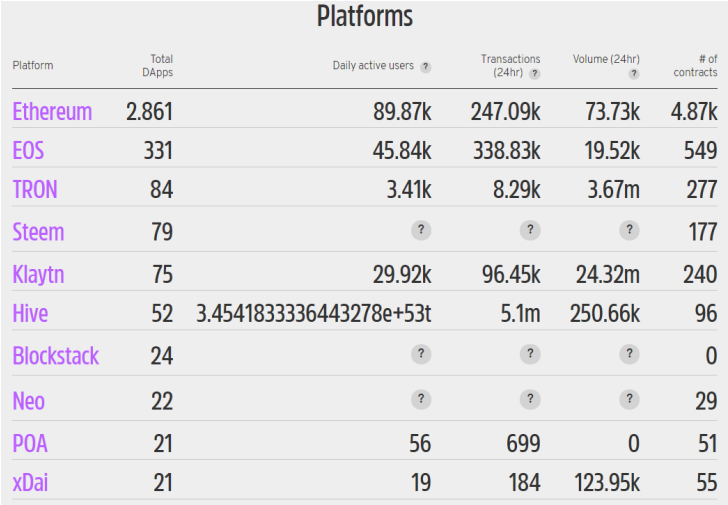
which extends the functionality of a single contract, the only limitation is that smart contracts can't send HTTP requests and therefore cannot interact with other online services, this feature is designed due to security concerns, directly opening HTTP requests may result in wrong or hacked responses and this is why oracles are created in order to make the Ehtereum blockchain aware of reliable data sources like Ether price versus US dollar. Chainlink and MakerDAO are two blockchain project which provide real time Ethereum price versus US dollar, they are decentralized oracles that publish real world data, in this case price data, on Ethereum blockchain.

An interesting development of the smart contracts is the Decentralized applications (Dapps). It is a combination of User interface and a smart contract used as back end, this combination effectively introduce a new kind of applications with following benefits:

- Zero downtime – once deployed smart contracts as back-end can't be shut-down by anyone, as long as the Ethereum blockchain is up, attacks like denial of service is virtually impossible.
- Privacy – only a wallet address is provided to use the Dapp, meaning that without any association of personal identity to Ethereum account it is impossible to figure out any personal information.
- Resistance to censorship – Dapps'operations can't be blocked by any single entity although with a big number of nodes it is possible fork before the Dapp creation.
- Complete data integrity – intrinsic to the blockchain, data stored inside are public and immutable.
- Trustless computation/verifiable behavior – smart contracts can be audited by anyone and the functioning is guaranteed by the source code.

From the website https://www.stateofthedapps.com/stats currently most of the Dapps are created on the Ethereum blockchain, even if other blockchains with smart contract capabilities exist fig. 1.5.

An Example of a use case can provide a better understanding of the smart contract life-cycle: supposing a real estate entrepreneur, after multiple rounds of discussion, strike a deal with a client to build a skyscraper with a total of 10 floors, then a team of Software engineers code the agreement from natural language to solidity code. Once the creation of the smart contract is done, it can be tested and validated (the blockchain is immutable). Once



Platforms

Platform	Total DApps	Daily active users	Transactions (24hr)	Volume (24hr)	# of contracts
Ethereum	2.861	89.87k	247.09k	73.73k	4.87k
EOS	331	45.84k	338.83k	19.52k	549
TRON	84	3.41k	8.29k	3.67m	277
Steem	79				177
Klaytn	75	29.92k	96.45k	24.32m	240
Hive	52	3.4541833336443278e+53t	5.1m	250.66k	96
Blockstack	24				0
Neo	22				29
POA	21	56	699	0	51
xDai	21	19	184	123.95k	55

Figure 1.5: Dapps platform statistics.
In the top 10 blockchain for Dapps Ethereum surpass EOS blockchain (the second place) by nearly a factor of $\times 9$.

Phase	Challenge
Creation	Readability Functional issue
Deployment	Contract correctness Dynamic control flowc
Execution	Trustworthy oracle Transaction ordering Execution efficiency
Completion	Privacy and Security Scams

Table 1.1: Challenges in each phase of smart contract [6]

deployed, involved parties sign the contract with their respective wallet, and the corresponding assets are locked for the entire duration of the deal.

The execution takes place in the blockchain, once programmed conditions are met or better the contractual condition are satisfied then contractual assets follow the programmed transfers automatically performing transactions. With the updated state of the blockchain the assets of both parties are unlocked and transferred. In Zheng et al. [6] several problems arise during the life-cycle of the smart contract, the benefits with respect to traditional contracts are efficiency of the operation, locked assets in case of fraud (this way the client and supplier have interest to unlock the assets), always verifiable contract.

The vulnerabilities are also present in any human activity especially in this new form of agreement, here in table 1.1 some of the problems are reported.

1.2.3 Notable problems

Along the issues in table 1.1, the Solidity programming language has got some specific problems (listed also in the official documentation[7]):

1. Unchecked External Call - when performing external calls with *transfer*, *send* or directly *call* methods, the developer expect a revert to occur if the method fails but he doesn't check for the return value, e.g. `if(!addr.send(1)) {revert();}`.

2. Costly Loops - computational power are estimated with Gas and paid in Ethereum, infinite loops and inefficient array manipulation can result in exhaustion of all available Gas provided to the contract.

3. Overpowered Owner - when the contract functions can be called just by the owner for example creation of new tokens, the smart contracts are called overpowered and it lacks of trustless principle. This can be solved by voting functions.

4. Arithmetic Precision - Solidity doesn't support natively float divisions, e.g. divison of two integers 3 and 2 will result in 1 instead of 1.5. In any equation it is better to mutiply first then divide.

5. Relying on tx.origin - the caller is not always the first account in the call chain (tx.origin), therefore using tx.origin as authorization to transfer funds may result in attack from hackers and drained funds, e.g.

```
contract TxUserWallet {
    address owner;
    constructor() {
        owner = msg.sender;
    }
    function transferTo(address payable dest, uint amount
    ) public {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Attack contract:

```
interface TxUserWallet {
    function transferTo(address payable dest, uint amount
    ) external;
```

```
}
contract TxAttackWallet {
    address payable owner;
    constructor() {
        owner = payable(msg.sender);
    }
    receive() external payable {
        TxUserWallet(msg.sender).transferTo(owner,
        msg.sender.balance);
    }
}
```

if the attacker use a contract with calls to bugged function trasferTo() then the vecification is based on tx.origin (the original external account that started the transaction) instead of msg.sender (the immediate account that invokes the function), then the attacked contract verify and all the funds are drained.

6. Overflow / Underflow - limited memory size 256 bits in EVM imply the use of "SafeMath" library to throw after performing wrong operations.

7. Unsafe Type Inference - Var declaration may result in unexpected type dimension, but in Solidity v0.7.0 Var is removed and not supported anymore.

8. Improper Transfer - using send() instead of transfer() may result in burned tokens when the transaction is unsuccessful.

9. In-Loop Transfers - transfer() method used inside a loop is dangerous because one exception can cause the whole transaction to fail and reverted.

10. Timestamp dependence - the blockchain doesn't provide a global time instead smart contracts can only use block.timestamp which is unreliable and under control of miners.

In the brief history of Ethereum the most notable attack is "The DAO" attack, reported also on Bloomberg [8], the attack exploited loopholes inside The DAO smart contract and gained 50 million dollars worth of ethereum. DAO is an acronym of Decentralized Autonomous Organization, generally they distributes tokens to participants and unlock funds by voting on project proposals like MakerDAO. The purpose of "The DAO" instead was to fund Dapps and grow the Ethereum ecosystem, it was created through a crowd-sale of tokens and the total raised funds were 150 million US dollars worth of Ethereum. The smart contract of "The DAO" contained a loophole within

the DAO split function, the function was created to withdraw the profits token holders made [8], but it didn't accounted for recursive withdraw.

First of all when DAO token holders make a proposal to give funds (in Ether) to a certain project, the proposal will get voted. If the 20% of the quorum vote positively the proposal pass and a smart contract containing the funds will be sent to a "startup" Dapp, but in order to protect the minority who vote negatively and don't want their funds to transfer to a project they don't believe then they can call a split of the DAO creating a Child DAO and withdraw the funds inside it. The Child DAO contains the equivalent Ether of the DAO token received from the crowd sale plus the portion of the profit that The DAO generated for the token holder.
Here are the codes to sum up the vulnerability fig. 1.6 fig. 1.7 fig. 1.8:

```
function splitDAO(uint _proposalID, address _newCurator) noEther onlyTokenholders returns (bool _success){
    // Move ether and assign new Tokens
    uint fundsToBeMoved = (balances[msg.sender] * p.splitData[0].splitBalance) / p.splitData[0].totalSupply;
    if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == false)
        throw;
    // Burn DAO Tokens
    Transfer(msg.sender, 0, balances[msg.sender]);
    withdrawRewardFor(msg.sender); // be nice, and get his rewards
    totalSupply -= balances[msg.sender];
    balances[msg.sender] = 0;
    paidOut[msg.sender] = 0;
    return true;
}
```

Figure 1.6: SplitDao function code

```
function withdrawRewardFor(address _account) noEther internal returns (bool _success) {
    if ((balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply < paidOut[_account])
        throw;
    uint reward = (balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply - paidOut[_account];
    if (!rewardAccount.payOut(_account, reward))
        throw;
    paidOut[_account] += reward;
    return true;
}
```

Figure 1.7: Withdraw function code

The splitDAO() function contains a vulnerability and the hacker successfully made an attack using a Proxy contract which call splitDAO() again and again before the function update its balance: the vulnerability is inside the withdrawRewardFor() function that call the payOut() which trigger a fallback function inside the Proxy contract and of course inside the Proxy contract the splitDAO() is called again making a loop where balance is never updated [9].

```
function payOut(address _recipient, uint _amount) returns (bool) {
    if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient != owner))
        throw;
    if (_recipient.call.value(_amount)()) {
        PayOut(_recipient, _amount);
        return true;
    } else {
        return false;
    }
}
```

Figure 1.8: Payout function code

The attack makes withdraw from The DAO contract a re-entrancy problem, meaning withdrawing more than once the real amount of tokens.

After The DAO attack, the initial response of the Ethereum founder Vitalik Buterin was to initiate a soft fork by adding a code snippets to blacklist the attacker and making stolen funds impossible to move, the alternative was a hard fork which effectively rolls back the previously mined block containing The Dao attack. Fork means that the network participants collectively follow new common rules upgrading the client containing the EVM. Not all the network participants agreed to this fork and so two parallel blockchains were created: Ethereum with new rules and Ethereum Classic with the old rules.

The Ethereum Classic community strongly believed that blockchain should be immutable and an error in the smart contract must be payed as it is: this is also the position of the hacker who claimed that he rightfully claimed the rewards. The majority of the network still considered The DAO attack too big and also the team behind the DAO contract supported the hard fork and subsequently the funds were moved to a recovery address where they could be exchanged back to Ethereum by their original owners (not in the Ethereum Classic blockchain). The DAO token was delisted from major exchanges. This event became a pillar in the early days of the Ethereum blockchain and since then researches, focused to avoid this problem, were made.

The smart contracts may contains intended or non intended vulnerabilities, problems like re-entrancy which caused The DAO hack can be found using tools like Slither and Mythril [7] but other common bugs in general can be prevented with better code documentation and tools that enable this process. The goal of this thesis is to support the programmers, beginners and eventually non programmers to understand and read a source code file of any smart contract.

1.2.4 The current state of researches

Research on problems cited in section 1.2.3 and table 1.1 have already made progress: regarding the readability problems and creation phase, which are the main focus of this thesis, many approaches have been suggested. Erays [10] is one of the tool created to convert compiled code into pseudo code in order to help auditors to verify smart contracts, in the research of Zhou et al. [10] they found that around 70% of contracts source code are no released and these smart contracts involves 30% of transactions in the network. ADICO grammar proposed by Frantz et al. [11] provide a framework to semi translate Natural language to ADICO component, ADICO stays for Attribute (actor's characteristics and attributes), Deontic (nature of statements, obligation, permission), aIm (action and outcome of this statement regulates), Conditions (contextual conditions), Or else (consequences). From table 1.2 it is possible to write an example: People (A) must (D) vote (I) every four years (C), or else they face a fine (O).

ADICO component	Solidity contruct
Attribute	Structs
Deontic	Function modifiers
aIm	Funcitons, Events
Conditions	If statements
Or else	throws/alternatives

Table 1.2: Corresponding ADICO components in solidity

ADICO can help the user to code a smart contracts transforming natural language into a intermediate representations then it can be mapped in Solidity, this process can be also nested. In the same direction a considerable amount of researches provides translation from natural language to pseudo-code or logic representations ready to be used by programmers to code in solidity language, e.g. tools capable of information retrieval [12] as data extraction method and then through templates custom smart contracts are created, another paper (Clack,2016) [13] propose to add also a legal prose inside the smart contracts templates making de-facto smart contracts legal electronic documents which can be consulted in any circumstances. Other efforts to help solidity programmers are represented by Mao et al. [14] and Guida et al. [15], the former creates visual programming tool by extracting commonly used functions in solidity source code and create drag and drop custom blocks while the latter also use visual programming but in a service oriented way, which is to retrieve and re-use third party contracts stored inside a registry

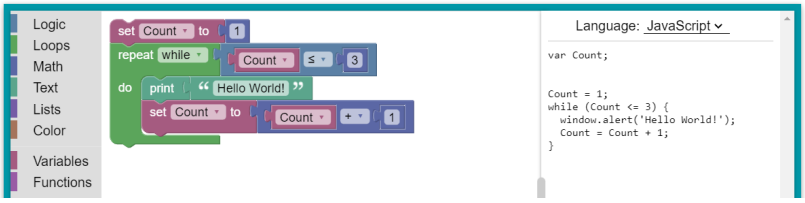


Figure 1.9: Example of visual programming

and use visual editor to compose the actual smart contract fig. 1.9. Visual programming may help also non programmers to create smart contracts.

The opposite direction regarding readability is to convert source code to natural language. Attempts like Regnath et al. [16], Li et al. [17] represents the current state of the art in solidity translation into natural language, although they are drastically different. Regnath et al. created SmaCoNat (SMArt COntract in NATural language) which is a domain specific language, it tries to directly write smart contracts in a more human readable syntax, the development of SmaCoNat is carried out with Xtext, a domain specific language framework. The readability of smart contracts is greatly improved this way as shown in fig. 1.10

```
Contract in SmaCoNat version 0.1.

§ Involved Accounts:
Account 'BarrierIn' by 'AComp' by Genesis alias 'BarrierIn'.
Account 'BarrierOut' by 'AComp' by Genesis alias 'BarrierOut'.

§ Involved Assets:
Asset 'TheCoin' by Genesis alias 'TheCoin'.
Asset 'ParkTicket' by Self alias 'Ticket'.
Asset 'OpenBarrier' by Self alias 'Open'.

§ Agreement:
Self issues 'Ticket' with value 42.
Self issues 'Open' with value 1.

§ Input Event:
if Input is equal to 'TheCoin' from Anyone
and if value of Input is equal to 0.3
then
   Self transfers 'Ticket' with value 1 to owner of Input.
   Self transfers 'Open' with value 1 to 'BarrierIn'.
   Self issues 'Open' with value 1.
endif

if Input is equal to 'Ticket' from Anyone then
   Self transfers 'Open' with value 1 to 'BarrierOut'.
   Self issues 'Open' with value 1.
endif
```

Figure 1.10: SmaCoNat example

Li et al. [17] instead use a more traditional approach using custom NLP tools to extract important data from smart contracts and then use templates to create natural language summary of solidity functions, Li et al. [17] currently represents the state of the art in solidity code summarization. The direction taken by Li et al. have still big improvement room using state of the arts techniques from computer science field.

1.3 SolidparticSummarizer

Summary of functions in source code can be directly taken from good quality comments written above function codes, this is why in this thesis summary generation and comment generation task are considered the same under readability point of view. With this in mind the thesis propose a new code summarization system for solidity source code called SolidityParticSummarizer.

SolidityParticSummarizer includes a custom set of scripts to collect and process raw Solidity smart contracts inside the SmartContractDatabase repository and the fork of a Transformer implementation to generate Solidity code summaries.

Chapter 2

State of the Art

The first paper to research direct translation from Solidity programming language to natural language is *"Towards Interpreting Smart Contract against Contract Fraud: A Practical and Automatic Realization"*(Li, 2020) [17]. Its aim is to enable people without computer science background to understand and operate Ethereum smart contracts, this way there is a possibility to reduce fraud while operating with smart contracts.

The author of the paper (Li, 2020) [17] built a novel tool named SMTranslator to automatically generate readable document from smart contracts. This tool is not published yet but according to the paper it transforms the smart contracts into structured files identifying functions, then using custom natural language generation techniques, it generates documents that can describe correctly the smart contract. There is a similar field of study in the broad coding world, that is called Code Summarization, but in this case the target audience is of course code maintainers and people with basic programming skills.

The following sections presents the tool SMTranslator, the research and advances in Code Summarization field, the Transformer architecture and the metrics used in NLP.

2.1 Smart Contract Translator

The paper of SMTranslator (Smart Contract Translator) [17] takes into consideration the following hypothesis on the reason why code summarization may fail to translate smart contracts:

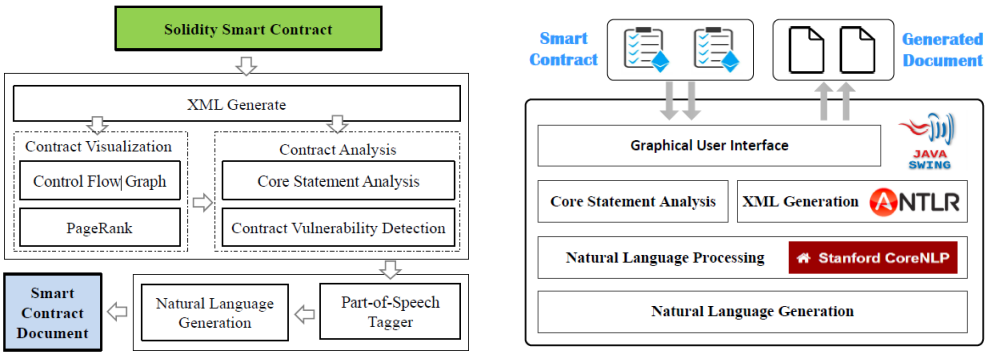
1. The code summarization is designed for developers who have programming skills and not individuals without computer science background,

an example of the final user is the lawyer.

2. The summarization schemes summarize the meaning of a function according to function name but according to the observation of the authors the function name may not correspond to the code meaning.

3. The research on documentation generation can't discover vulnerabilities and present them.

The list item 1, item 2 and item 3 are legitimate reasons for not using code summarization, cases of fraud has already happened and it still happens with new types of smart contract table 1.1 and new token sales as described in section 1.2.3, however the creation of a smart contract always involves a bona fide programmer so it is reasonable to create a tool for programmers instead of general public. The second hypothesis list item 2 also isn't true entirely because recent advances in Code Summarization doesn't take into account only the function *verb* or the function name anymore but there is a wide range of options as described in the next section 2.2.



(a) The architecture of SMTranslator (b) The system design of SMTranslator

Figure 2.1: SMTranslator overview with each component

Based on the paper of the authors (Li, 2020) [17] the architecture of SMTranslator is as shown in fig. 2.1, the generated smart contract document should contain 6 or less statements according to the template. The statements are:

- short description of the function method like a summary;
- return description on type and value;
- modifier description which is special conditions that the method have to satisfy;

- input description on the inserted parameters;
- core description which is the main statement;
- call description if the method have call functions;

A practical example can be synthesized in the pair: *"function receiveEther() returns(bool)"* as input and *"This method receiveEther() can be used to receive Ether and returns bool value"*. This simple example already contradict the paper hypothesis because the output is the same as a code summarization tool would do.

The evaluation on the tool is performed using a questionnaire, it use score from "Strongly agree","Agree","Disagree" and "Strongly disagree". The questions are:

- Q1 Usability, The tool is easy to use and operate.
- Q2 Accuracy, The explanations and summaries for a method are accurate.
- Q3 Readability The generated summaries are easy to read and I can totally understand the meaning of each generated sentences.
- Q4 Conciseness The summaries do not contain unnecessary information.
- Q5 Instructiveness I can easily use a specific method under the direction of the explanations.
- Q6 Core Analysis I feel the tool for core statement analysis of a method is accurate and does not miss important information.

Ten participants with basic blockchain knowledge answered the questionnaire and their answer are summarized in table 2.1.

Question	Strongly agree	Agree	Disagree	Strongly disagree
Q1	3/10	5/10	2/10	0
Q2	0	6/10	4/10	0
Q3	9/10	1/10	0	0
Q4	9/10	1/10	0	0
Q5	0	3/4	1/4	0
Q6	0	64%	23%	13%

Table 2.1: Questions asked in the questionnaire, and students opinions.

The Q1 to Q4 are questions regarding all the methods and generated documentations, 3 out of 10 participants rated strongly agree and so on, the Q5 is intended only for computer science students and they are 4 in total while the Q6 is also intended for the computer science students but each participant rated 50 randomly chosen methods and the results should be interpreted as the percentage on all the 200 ratings.

The SMTranslator results are up to now the state of the art for solidity language documentation generation although it starts with wrong assumptions, future projects need at least take it into consideration the paper background motivation. An other weakness of the paper is that the evaluation is based on an empirical questionnaire which is not objectively verifiable.

2.2 Code Summarization

According to statistical measurements, developers spend 59% of their time in code understanding, and this percentage is increasing over time due to the increasing complexity of complex software systems [18], this is the reason why during software maintenance and development, the readability and comprehension of program code is key to success for any programmer. Comments are essential to understand code better; in new programming languages such as Solidity, the comments represent an indispensable tool for newcomers and programmers who wants to add a new language inside their portfolio. Code summarization (CS) is the field that study the techniques used for getting information from source code.

The SMTranslator is de-facto a CS tool, it doesn't generate sentences that exceed high quality comments which a good programmer would write. In this section some of the CS methods are explained in order to categorize SMTranslator. New CS methods are introduced in order to solve the Comment generation problem which affects most of source code in the Solidity Smart Contracts ecosystem.

Until now according to Zhu et al. [19] most researches during 2010 to 2019 on CS can be divided in the technique used to extract information from source code, the generation of comments, the evaluation of the results and source code artifacts from which it is used to generate descriptions. The total number of evaluated papers by [19] is 41, they are the most relevant papers based on Google Scholar data, all of them are taken from the following digital libraries:

- ACM digital library.

- IEEE Explore.
- Science Direct.
- Springer Link.

2.2.1 Data Extraction

The table 2.2 shows the methods used to extract information from source code; most of them are based on Information Retrieval and Machine Learning techniques, before describing these methods, it is appropriate to dive into what compose a source code. Source Code usually are text file written in programming languages and composed of different parts, the most useful parts of the source code in CS are comments and code, different CS techniques use different kinds of code artefacts, that ranges from the whole document to functions and single variables, obtaining different results.

Method	Frequency	Rate
Information Retrieval	17	41%
Machine Learning	13	32%
Stereotype Identification	8	20%
Natural Language Process	7	17%
External Description Usage	4	10%

Table 2.2: CS data extraction methods collected from 2010 to 2019.

Here there is a general description of the extraction methods:

- **Information retrieval** : it extract information based on the call graph representation of the source code and assign the call graph to a topic model. The topic model is a cluster of similar words build to to predict the class of the comments.
- **Machine Learning and Artificial Neural Network** : ML and ANN are divided into two classes: supervised learning and unsupervised learning. Both supervised and unsupervised learning are data dependent which means that these models learn from the training input data. The difference between ML and all other data extraction methods is that ML can be generalized to understand any programming language, while classic data extraction methods must be configured to specific use and then applied.

- **Stereotype Identification** : it categorize methods based on its function identifying the stereotype of the method, e.g. a method whose responsibility is to build a class is labelled as a constructor. With the creation of a set of finite stereotypes, it is possible to define a template for the extracted words.

- **Natural Language Process** : NLP is different from previous methods. It analyzes natural language, exploits the linguistic relationship between words, and then uses this information for automatic comment generation, usually creating word tagging techniques based on grammar.

- **External Description Usage** : it uses external information from well known websites such as StackOverflow, Github to extract code-description mappings or existing software repositories to detect similar codes and their summary, in this case some NLP post process mapping must be used to find most relevant information.

Data extraction can be seen as a encoder of the source code, which are most of the method used in CS, even though with different techniques, they carry out the same function, encoding means an intermediate information are stored and ready to be used in next steps.

2.2.2 Summary Generation

The comment generation, or better the summary generation described in table 2.3 shows the dominant research are template-based description of the source code; this fact is attributed to the timing of important discoveries in NLP. Machine learning applied to Natural language began to develop in the new millennia with first probabilistic models, then the Neural language models with artificial neural networks and finally deep learning models. Meanwhile, the classic Natural Language Processing is heavily based on linguistic and grammar theories, which is limited in generation tasks.

Method	Frequency	Rate
Template based	19	46%
Machine learning based	12	29%
Term based	7	17%
External Description based	4	10%

Table 2.3: CS summary generation methods collected from 2010 to 2019.

- **Template-based** summarization is the most common natural language summary generation method. In template-based summarization, researchers predefine a set of summary templates and fill in the templates based on the type of the target code segment and other information such as interfaces, packages and parameters. Template based summarization is still relevant in the Machine Learning and Neural Network era because it works with any intermediate extracted information.

- **Machine learning** based summary generation is almost always paired with ML data extraction methods; this way, it forms an encoder-decoder paradigm that creates a model based on the training dataset. The early models were based on statistical machine translation. While after the rapid development of neural networks in general, the sequence to sequence models emerged and at present, we have attention-based models, which are the state of the art of neural machine translation. translation while after the rapid development of neural networks in general, the sequence to sequence models emerged and at present we have attention based models which are the state of the art of neural machine translation.

- **Term-based summarization** is generation of a summary that contains the most relevant terms for a specific software code snippets or code artefact (functions, declarations, loops, ecc...). Most of term-based methods use information retrieval techniques to extract information and then generate a keyword list from it in different approaches. The generated keyword list captures source code semantics on which developers focus most of their attention.

- **External Description** based method uses external data such as comment code mappings in other repositories or website forums. It's a big data effort, it maps code segment with comment segment found in online code repositories and it usually generate the summary of code with minimal change based on relevant comments. It is usually combined with same technique for data extraction. Inside a coding forum codes and descriptions are usually paired as questions and answers

2.2.3 Evaluation of CS techniques

CS techniques have different objectives, the main objective is to help programmers but there are different methods to help and different shading to obtain this objective, generally, the evaluation of CS techniques are categorized as

online, offline, human involved or not, intrinsic or extrinsic. The table 2.4 show the distribution of the evaluation methods:

Method	Frequency	Rate
Manual Evaluation	23	56%
Statistical Analysis	16	39%
Gold Standard Summary	7	17%
Extrinsic Evaluation	5	12%
None	4	10%

Table 2.4: CS summary generation methods collected from 2010 to 2019.

- **Manual Evaluation** is the most commonly used evaluation technique, it can be combined with other techniques, it dominates the evaluation scenery because if a group of expert programmers is found online or offline, their judgement corresponds to the final user judgement, which is a direct feedback. The participants can use "strongly agree" to "strongly disagree" with different intermediate levels to rate the generated documentation. Some evaluations choose to eliminate intermediate levels and avoid non-committal answers.

- **Statistical Analysis** includes statistical measurements such as precision, recall, F-score, machine translation scores. Statistical Analysis is based on the comparison between a baseline and the generated comments. Accuracy, precision, recall, and F-score are efficient ways to quantitatively evaluate the performance of natural language summary generator, they will be explained later in section 2.4. In artificial neural network studies, the most used metrics are Machine translation metrics such as BLEU(Bilingual Evaluation Understudy) score, METEOR (Metric for Evaluation of Translation with Explicit ORdering) and ROGUE (Recall-Oriented Understudy for Gisting Evaluation), this is because with an objective score the feedback loop in model training can be formed.

- **Gold Standard** Summary refers to human created summary as reference, the generated keywords list is compared to the human created to obtain the quality of the results. This method is often used by researchers in combination with manual evaluation, this way after comparing the generated comment and the gold standard one the person who is asked to evaluate may give a more accurate feedback.

- **Extrinsic Evaluation** is based on extrinsic strategies to evaluate the influence of the generated summaries on the reader, if the generated comments improve the reader's ability to program, such as increasing productivity, comprehension, typing speed. The evaluator usually is the researcher who observe online or offline the readers, in this case, readers become part of the evaluation process as they carry out the experiment by typing, by programming standard piece of algorithms.

2.2.4 Target Code Artefact

The last category for CS researches is the code artefact. There is a difference in summarizing the whole story with respect to just one paragraph, this is the concept behind the code artefact. Summarizing small pieces of code can drastically improve the comment generation, that's why the majority of researches are focused on specific code artefact, in table 2.5 there is the distribution of researches with respect to the chosen code artefact.

Code Artifact	Frequency	Rate
Method	13	56%
Code segment	12	39%
Code change	9	17%
Class	5	12%
Test case	2	10%
Package	1	2%
Variable	1	2%

Table 2.5: CS targets distribution.

- **Methods and functions** are the most popular choice among code artefacts, in this case usually comments above the methods plus the methods itself can provide datasets containing pairwise relationships.
- **Code segments** are code written on notorious websites such as Stack Overflow and GitHub, which include programmers comments explaining them, they vary in length.
- **Code change** are part of software versioning systems, whenever a portion of code is changed usually the programmer include a commit message along with the changed code, this way a natural association is created.

- **Class usually** is the target for power-full Object Oriented Programming languages like JAVA, the CS method in this case can explain the contents, responsibility and role of a particular Class.
- **Test case documentation** are researches focused on generating summaries for unit testing, this is a very specific target and difficult to generalize on the whole source code.
- **Package summarization** provides the summary for specific packages on their provided services, in this case it refers to the Java context.
- **Variable summarization** is focused on the parameters and different data types, it is never used alone because it can only be part of a larger summary like method summary.

Once Provided a general overview on the current research landscape and defined different possible directions of the thesis it is clear that to solve the problem described in the introduction a detailed choice must be made. First of all SMTranslator can be defined with CS categories and then a evaluation on the advantages and disadvantage can be made.

2.2.5 Evaluation of SMTranslator

Under the framework given by Zhu et al. [19] the SMTranslator can be categorized as a CS tool using **information retrieval** plus **natural language processing** as data extraction method and **template based** summarization as summary generation method. In the SMTranslator **manual evaluation** is used as final evaluation because of a specific goal, which is helping non programmers to read smart contracts. Finally as code artefact the **Solidity functions** were used as target to generate their documentation.

The data extraction used in SMTranslator at best of its ability it can extract limited information, that is smaller or equal to the information needed for the template based summarization. A example of the template is provided in table 2.6.

Type	Template
Short_Description	"This method $< verb - object >$ and"
Return_Description	"Return $< statement >$"
Modifier_Description	"This method can be called if $< condition >$"
Input_Description	"The inputs are <params>"
Core_Statement	"$< main - action >$"
Call_Description	"This method is called by $< function >$"

Table 2.6: Example Template of SMTranslator.

Given a function like:

```
function add ( uint256 a , uint256 b )
    internal pure returns ( uint256 ) {
    uint256 c = a + b ;
    require (c > a , Safe Math:addition overflow );
    return c ;
}
```

The generated description is:
This method add two unsigned integers and return c. This method can only be called if *None*. The inputs are uint256 a and uint256b. The funciton add two numbers, throw on overflow. This method is called by addTokenTo.

One of the possible Gold standards:
/***
@dev Returns the addition of two unsigned integers, reverting on overflow. Counterpart to Solidity's "+" operator. Requirements: Addition cannot overflow.
*/

The Gold standard is the human written comment, which can slightly differ one from another, but it contains high-quality documentation of the function. At first glance, SMTranslator seems to be better than human-generated comments, but the core statement has a 13,6% of strongly negative mark on the generated core documentation. Because not every Solidity method is as simple as an addition, most of the time, the name of the function does not describe the function itself, especially for programmers who do not follow best practices. Another crucial flaw of the SMTranslator is the generation of the description without human written comments, Core_Statement analysis of pure source code fails in two cases when the source code is too long failing the selection of the main action and when there is no human comments to take as reference. These problems regard all tools that use template-based

summarization, templates limit summary generation, it is also true that with a large number of templates, it is possible to achieve outstanding results, but it is concentrated only in specific conditions. Lastly, the Solidity language is a relatively new language, the current version is 0.8.7, it has not reached the 1.0.0 version yet, which means that keywords and overall programming language are subject to change if SMTranslator is used in 2021 its performance may decrease and long configurations are required.

One of the possible solutions proposed by this thesis is to use novel approaches to CS, the current Machine Learning techniques and Neural Networks models are on the rise and they can potentially solve more generalized CS problems, which is why this thesis proposes the use of state of the art Neural Network model applied on Solidity Code Summarization problem. SolidparticSummarizer is proposed as a solution to help non programmers or programmers with basic knowledge of Solidity to improve and speed up the understanding of source code, for this purpose, CS is the central part of this thesis given that most smart contract doesn't provide documentation.

SolidpartySummarizer, as described in section 1.3, is composed by SmartContractDatabase and the fork NeuroCodeSum, while the former repository is a custom data collector and data parser, the latter is a Transformer based Neural Network model. It is a state of the art model which has never been applied to Solidity code yet. In the following sections, a detailed analysis of Transformer architecture is presented with a comparison with the previous Neural Network architectures.

2.3 Transformers

In recent years Natural Language Processing is more and more associated with AI since the advances NLP shifted from a heavily linguistic and probabilistic field to a more computational oriented approach, the linguistic influence or, in computational terms, the Rule-based system still influence the NLP researches. However, Machine learning-based systems can now outperform them in the same given task with better results.

First of all, Machine Learning is the study of algorithms that can improve with experience and data, it means that the algorithms must perform several cycles to achieve a good result (either exact or approximate result within the tolerance range), and data must be provided in order to achieve that. With this definition, Deep learning is a subset of Machine learning that is based on Neural Networks as the main framework. A complete training cycle over the whole dataset is called Epoch.

Neural Machine Translation (NMT) is machine translation using Neural Networks, the input of NMT is a sequence of symbols and the final output of the NMT is another sequence that best describes the original sequence. In natural language, the translation from one language to another is complicated, the context greatly influences the translation and this is one of the reasons why it is an absolutely challenging task for both humans and machines.

NMT is introduced because CS training models have got small differences with respect to translation models. They both need the same input and require an understanding of context, and they can both be studied as Encoder-Decoder models. In NMT, the state of the art models are all based on Transformers. Almost every time using statistical evaluations such as BLUE, Transformer based models tops the charts. For the comment generation task, SolidparticySummarizer employed a Transformer based model, which revealed pretty solid and consistent over most of the inputs.

2.3.1 Introduction

Transformer based model is a significant milestone in NLP. The original paper "Attention is All you need" (Vaswani, 2017) [20], opened a research field that includes Generative Pre-trained Transformer (GPT) and Bidirectional Encoder Representations from Transformers (BERT). These two models generated infinite other variations on their own. This thesis will not explore GPT and BERT, but they represent the cutting edge of NLP technology. At the start of this thesis, the Transformer was the most promising model for Code Summarization as [21] was just published, and it was a perfect tool for Solidity Summarization, future research may involve GPT and BERT applied on Solidity programming language.

Why does Transformer represent such a revolutionary idea? Furthermore, how NMT got to this point? In this brief introduction to Transformers, the following sections will try to explain them.

2.3.2 Embeddings

One of the first thing a person can ask is how can words be represented in ML models, in statistical models words are discrete and we can just assign its frequency inside a text, this is precisely the Bag of Words model. This model can be generalized with n-gram model where n stay for the number of words you chose to assign their frequency, for n equal to 1 it's the equivalent of Bag of Words.

The example sentence "I do not like pizza with pineapples but I like pizza",

in Bag of Words or Uni-gram model is shown in table 2.7a and Bi-gram model is shown in table 2.7b of the same table 2.7.

Uni-gram	frequency
I	2
do	1
not	1
like	2
pizza	2
with	1
pineapple	1
but	1

(a) Uni-gram example also called Bag of words.

Bi-gram	frequency
I do	1
do not	1
not like	1
like pizza	2
pizza with	1
with pineapple	1
pineapple but	1
but I	1
I like	1

(b) Bi-gram example

Table 2.7: The n-gram model example

N-gram models are powerful in Statistical Machine Learning algorithms and they are usually based on probability distribution and statistical analysis, one of the most important model is the Bayesian estimation.

In Neural Networks models, discrete representation of words are not good enough, another milestone of NLP indeed is the Word Embedding described in Word2Vec paper [22] which use both Continuous bag of words (CBOW) and skip-gram technique to produce a distributed representation of words, with each vector representing a word, forming a cloud of words. How it is possible to transform words into vectors? First of all the size of the vocabulary V and a learning dataset must be provided, then with one-hot encoding it is possible to encode each word, giving one out of V to the corresponding word and zero to all other elements table 2.8.

Vocabulary	0	1	2	3
king	1	0	0	0
queen	0	1	0	0
man	0	0	1	0
woman	0	0	0	1

Table 2.8: Example of one hot encoding.

Using the online tool provided by Rong et al. [23] it is possible to see the input layer x = $\{x_1, x_2, \ldots, x_V\}$, the input weight matrix W_{VxN}, the hidden

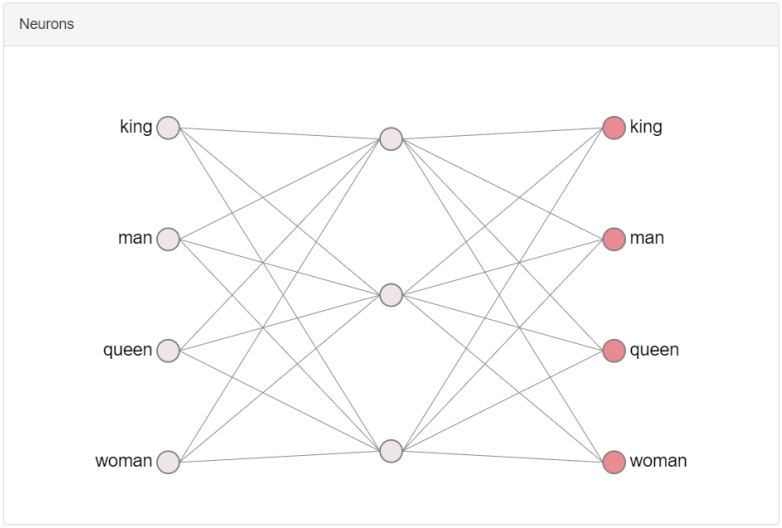


Figure 2.2: Word2Vec Neural Network

layer **h** with N features, the output weight matrix W'_{NxV} and the output layer y= $\{y_1, y_2, \ldots, y_V\}$ in fig. 2.2.

The word embeddings are just the vectors **v** connecting the input layer to the hidden layer, these values at first are randomly initialized, then using CBOW which means predicting the word given its context or using Skip-gram which means predicting the context given a word, it is possible to learn the vector values. In the example with table 2.8 vocabulary if we use CBOW and as input we give king and we want predict queen or we give input man and we want to predict woman, the training of the weights starts. With only one word as context fig. 2.3 is the Word2Vec neural network with layers fully connected. The hidden layer equation is eq. (2.1), where k is the position where $x_k = 1$.

$$\mathbf{h} = \mathbf{W}^T\mathbf{x} = \mathbf{W}^T_{(k,\cdot)} := \mathbf{v}^T_{w_I} \tag{2.1}$$

This is essentially copying the k-th row of W to h, where v_{wI} is the vector representation of the input word wI. Using the score **u** given by the output layer vectors $\mathbf{v}'^T_{w_j}$ eq. (2.2), it is possible to compute the multinomial distribution of the posterior distribution of words in our case is p(queen|king), this is essentially the softmax operation eq. (2.3).

$$u_j = \mathbf{v}'^T_{w_j}\mathbf{h} \tag{2.2}$$

$$p\left(w_j \mid w_I\right) = y_j = \frac{\exp\left(u_j\right)}{\sum_{j'=1}^{V} \exp\left(u_{j'}\right)} \tag{2.3}$$

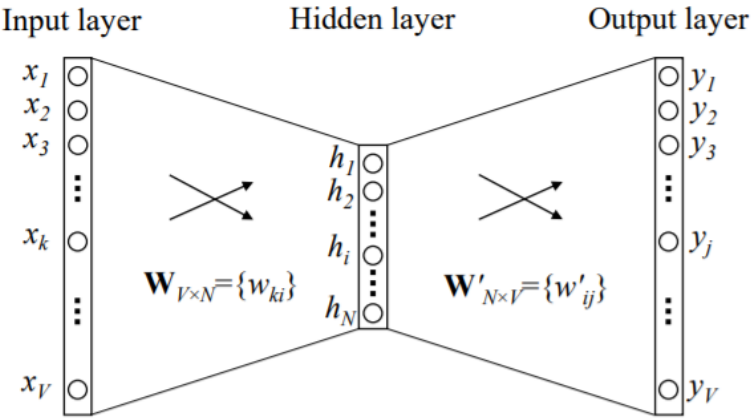


Figure 2.3: CBOW model with only one word in the context

where y_j is the output of the j-the unit in the output layer. The training objective is to maximize the probability of the output word $w_o = queen$ given the input word $w_I = king$ eq. (2.4). Denoting j^* index of w_O.

$$\begin{aligned} \max p\left(w_O \mid w_I\right) &= \max y_{j^*} \\ &= \max \log y_{j^*} \\ &= u_{j^*} - \log \sum_{j'=1}^{V} \exp\left(u_{j'}\right) := -E, \\ E &= -\log p\left(w_O \mid w_I\right) \qquad Lossfunction \end{aligned} \tag{2.4}$$

The loss function can be understood as a special case of the cross entropy function. Now with back propagation techniques it is possible to derive the update function for both output vectors and input vectors eq. (2.5).

$$\begin{aligned} \frac{\partial E}{\partial u_j} &= y_j - t_j := e_j \\ \frac{\partial E}{\partial w'_{ii}} &= \frac{\partial E}{\partial u_j} \cdot \frac{u_j}{\partial w'_{ii}} \\ \frac{\partial E}{\partial h_i} &= \sum_{j=1}^{V} \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} \\ \frac{\partial E}{\partial w_{ki}} &= \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_{ki}} \end{aligned} \tag{2.5}$$

The update functions for output vectors is eq. (2.6) and for the input vector

it is eq. (2.7):

$$\mathbf{v}'^{(\text{ new })}_{w_j} = \mathbf{v}'_{w_j}\ (\text{old})\ - \eta \cdot e_j \cdot \mathbf{h} \quad \text{for } j = 1, 2, \cdots, V \tag{2.6}$$

$$\text{EH}_i := \frac{\partial E}{\partial h_i} = \sum_{j=1}^{V} e_j \cdot w'_{ij} \quad , \quad \mathbf{v}^{(\text{new })}_{w_I} = \mathbf{v}^{(\text{old })}_{w_I} - \eta \cdot \text{EH} \tag{2.7}$$

In conclusion the Word Embeddings enabled the representation of words into

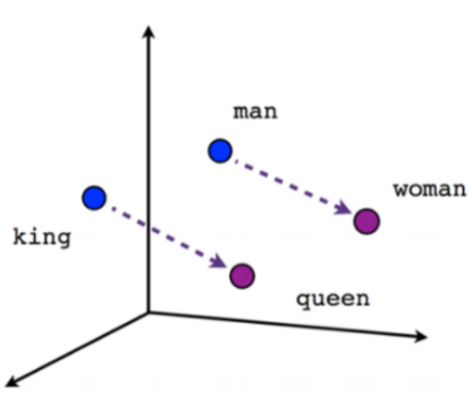


Figure 2.4: Vector representation of words,
in particular the male-female relationship.

vectors and they are fundamental for any Neural Machine Translation models, the embeddings obtained with the example vocabulary can be shown with the famous relation $king - man + woman = queen$ as described in fig. 2.4. Later on Word Embeddings techniques perform either explicit or implicit matrix factorization to word co-occurrence matrix.

2.3.3 The architecture

The Neural Machine Translation (NMT) in general (not only Attention-based models) has essentially two assumptions: the Embedding layer, which represent the continuous representation of words and a Neural Network framework for encoding and decoding sequences of symbols.
Recurrent Neural Networks(RNN) and Convolutional Neural Networks(CNN) are well established state of art in NMT before Transformer models but they have problems that limit their performances :

- Sequential processing: sentences must be processed words by words.

- Past information retained through past hidden states: sequential sequence to sequence models follow the Markov property, each state is assumed to be dependent only on the previously seen state.
- Parallel training: being sequential the training is slow and sentences can't be processed as whole, even creating Bi-directional model (making stronger hidden representation in both direction) the performance lacks behind Transformer.
- Vanishing and Exploding gradient: in RNN the training of long sequences may be biased on most recent inputs.
- Memory access: to learn contextual representations a model should access to older inputs, LSTM and RNN are not capable of long contextual understanding given a limited size of memory cell.

Transformer is a Sequence to Sequence based system, it encodes a input sequence x = $\{x_1, x_2, \ldots, x_n\}$ (n number of words) to an intermediate sequence z = $\{z_1, z_2, \ldots, z_n\}$, that in its turn is decoded to the final sequence y = $\{y_1, y_2, \ldots, y_m\}$. Notice that the final sequence may got different length than the input. In fig. 2.5 some smaller blocks have to be explained: the Attention, Multi-head Attention, Encoder-Decoder, Positional Encoding. They are the building blocks of the architecture.

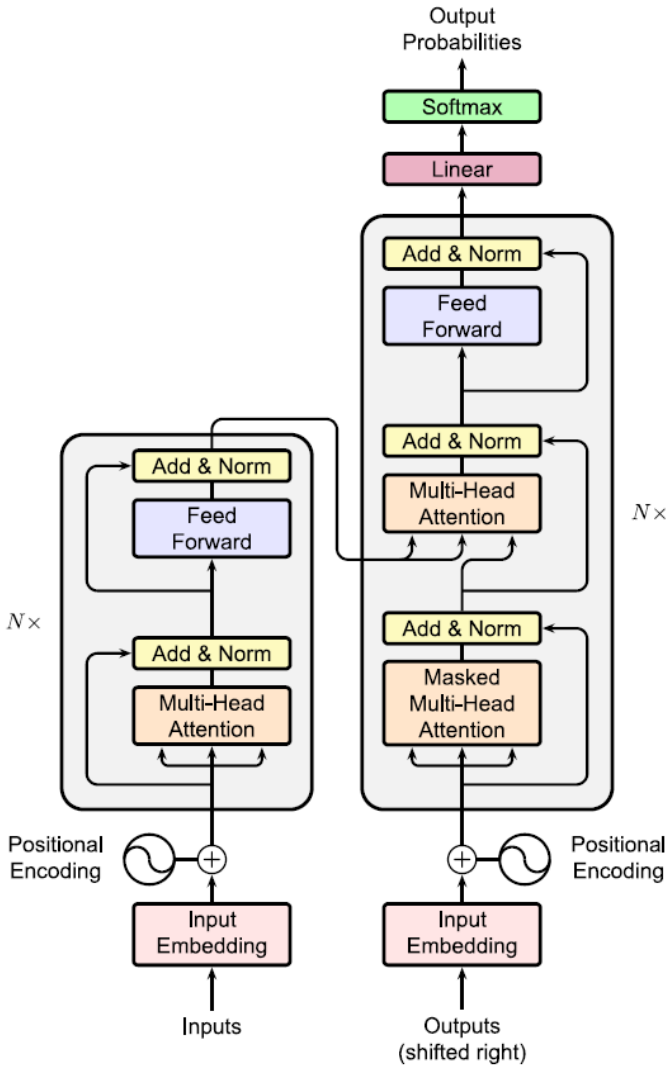
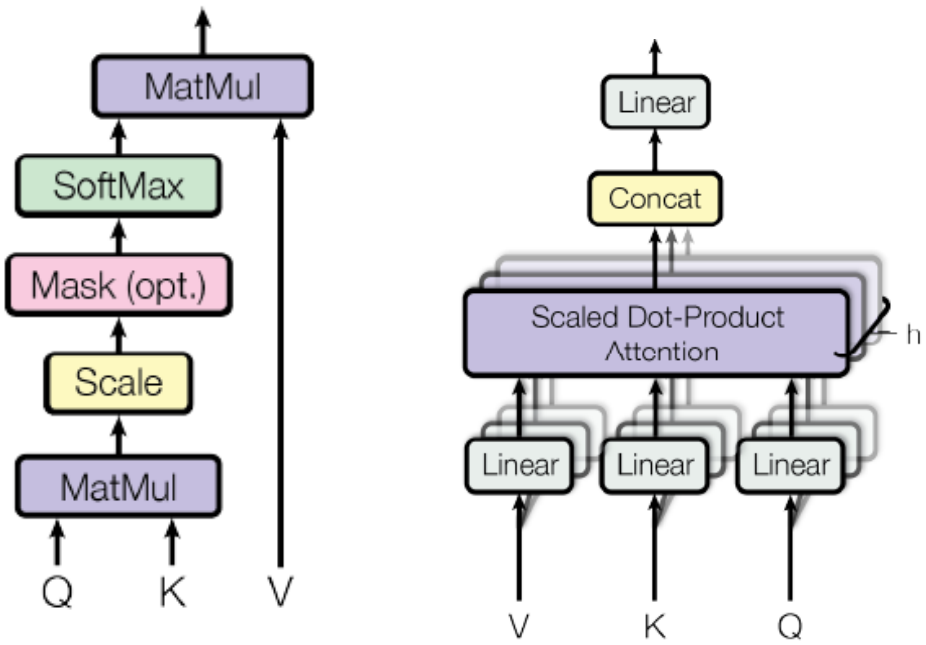


Figure 2.5: The transformer architecture

Attention

In particular the scaled dot product Attention is used in the original Transformer paper (Vaswani,2017) [20], in fig. 2.6a three matrices Q, K, V are used as input to compute attention values.



(a) Scaled Dot Product Attention (b) Multi-head Attention

Figure 2.6: Transformer Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.8}$$

Attention is computed as in eq. (2.8), the $Q(x_i)$ (query), $K(x_i)$ (key) and $V(x_i)$ (value) are Matrices respectively composed by $q_i = W_q \cdot x_i$, $k_i = W_k \cdot x_i$, $v_i = W_v \cdot x_i$, where W is a pre-trained embedding matrix. It is useful to notice that the encoder and decoder are different, while the encoder attention is focused only on the source symbols, the decoder attention is divided in the first stage by the first multi-head attention with target symbols as input, then as second multi-head attention with query and key from the encoder block and as value the intermediate representation produced from the first stage.

Intuitively the query is the word under examination, the set of keys are the vectors to compare against and finally the value is the best matched

vector. For encoders it is more precise to talk about Self Attention, meaning that $Q \tilde{=} K \tilde{=} V$ and the final product is the Self-Attention matrix. In the Attention matrix the value d_k is the dimension of the query and keys, d_k is equal to d_v in Self-Attention and different in the decoder Attention, d_v is the dimension of value vectors. The factor $\sqrt{d_k}$ is used in eq. (2.8) to help the softmax function in extreme small gradient regions.

Instead of producing only one Attention Matrix fig. 2.6b suggest to project linearly the Q,K,V matrix and perform attention h times, then the Attention matrix can be concatenated and then linearly projected. The Attention matrices are called attention heads eq. (2.9).

$$\begin{aligned} \text{MultiHead}(Q,K,V) &= \text{Concat}\ (\ \text{head}_{\ 1}, \ldots,\ \text{head}_{\ \text{h}})\, W^O \\ \text{where head}_{\ \text{i}} &= \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned} \tag{2.9}$$

The concatenation of heads have dimension $d_k \times d_{model}$ and the Weight matrix W^O has dimension $h \cdot d_k \times d_{model}$. The parameter d_k is actually equal to d_{model}/h.

Encoder stages

The Encoder have two stages the first stage is splitting the input embeddings $X = x_i$ into three different matrices as described in section 2.3.3 then perform the Multi-head self attention to obtain a intermediate $U = u_i$ matrix and finally adding it with the input creating a residual connection R. The residuals are then transformed into the final intermediate representation z which is the output of the encoder. The Encoder can be described mathematically with element wise notation of eq. (2.10), eq. (2.11), eq. (2.12), eq. (2.13), eq. (2.14).

$$\alpha_{i,j} = \text{softmax}_j\left(\frac{Q\left(\mathbf{x}_i\right) K^T\left(\mathbf{x}_j\right)}{\sqrt{d_k}}\right) \cdot V\left(\mathbf{x}_j\right) \tag{2.10}$$

$$\mathbf{u}_i = W^O \sum_{h=1}^{N} \sum_{j=1}^{n} \alpha_{i,j}^{(h)}, \qquad W^O \in \mathbb{R}^{h \cdot d_k \times d_{model}} \tag{2.11}$$

$$\mathbf{r}_i = \text{LayerNorm}\left(\mathbf{x}_i + \mathbf{u}_i\right) \tag{2.12}$$

$$\mathbf{z}_i' = W_2^T\, \text{ReLU}\left(W_1^T \mathbf{r}_i\right), \qquad W_1 \in \mathbb{R}^{d_{model} \times n}, W_2 \in \mathbb{R}^{n \times d_{model}} \tag{2.13}$$

$$\mathbf{z}_i = \text{LayerNorm}\left(\mathbf{r}_i + \mathbf{z}_i'\right) \tag{2.14}$$

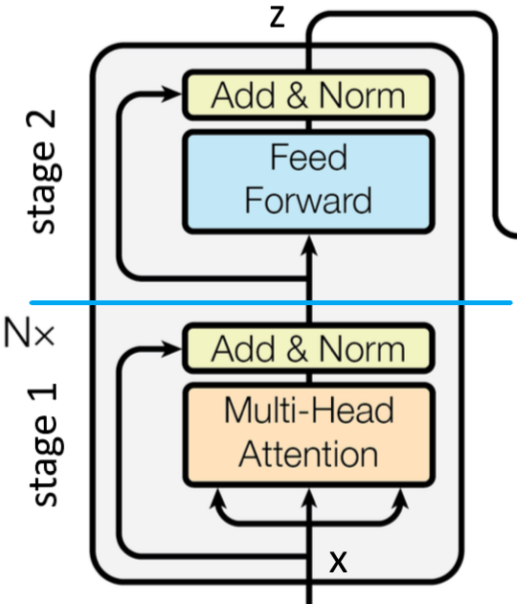


Figure 2.7: Stages in Encoder block

The Transformer encoder architecture fig. 2.7 is a stack of of N identical layer. The first stage is described by eq. (2.10), eq. (2.11), eq. (2.12) and the second stage is a fully connected feed forward network (FFN) eq. (2.13) plus the layer normalization eq. (2.14). In each stage there is a residual connection [24]. The residual connection is a type of learning based on the residual function, intuitively given $H(x)$ final mapping of x after any two layers, and F(x) the output of the first layer, the final mapping should be equal to $H(x) = F(x) + x$ fig. 2.8. The dimensions of the embeddings are all equal to $d_{model} = 512$, in detail $x_i \in \mathbb{R}^{1 \times d_{model}}$, $Q \in \mathbb{R}^{n \times d_{model}}$, $A \in \mathbb{R}^{n \times d_{model}}$, $U \in \mathbb{R}^{h \cdot d_k \times d_{model}}$, $Z \in \mathbb{R}^{n \times d_{model}}$. The Feed Forward Network (FFN) is just a projection of the square matrix into a intermediate representation, a visual representation of the input with respect to the intermediate representation is fig. 2.9, where self attention of each layer is different and each attention head learns different features.

Decoder stages

The decoder also is composed by N identical layers. The first stage is equivalent to the encoder but as input it has the target embeddings fig. 2.10, the second stage has got Q and K matrices from the encoder as input (Z) and finally the third stage is again a FFN sub-layer employing residual connections followed by $LayerNormalization()$ of the previous sub-layer output and the skipped

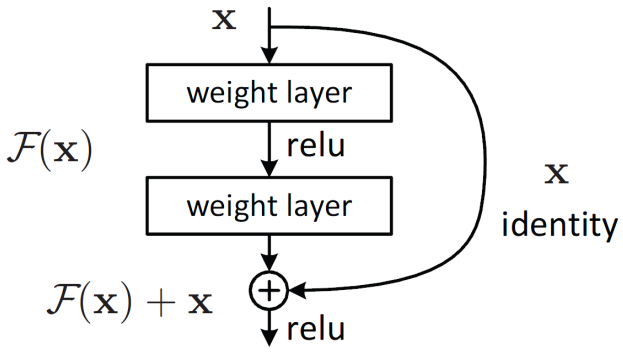


Figure 2.8: Residual connection H(x) equal to F(x)+x

input. Actually the input of the decoder is a little different than that of the encoder, the input embeddings are masked and shifted. The shifting on the input of the decoder ensures that at position i, the predictions are only based on token with position less than i, while the mask are used to ensure that during the training the word can't attend future words in advance. Decoding involves multiple steps, at first step the <Start> token is used to predict the target embedding then loss is computed and weights are updated then given the first token and the second token (which is the first target word) the second prediction is made and so on until the <End> token of the encoder.

Positional Encoding

In addition to learned input Embedding a Positional encoding is added to the encoder/decoder input, in this way the relative and absolute positional information is not lost, the input sequence of words are not always fixed and so position and dimensionality matters. The *pos* is the position of the word inside a sentence and i is the index inside the embedding vector.

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_{\text{model}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{\text{model}}}\right) \end{aligned} \tag{2.15}$$

Final layer

The output of the decoder is actually still a intermediate representation y' and the final linear layer with n fully connected neurons one for each word will sum the embeddings weights and compute the softmax of the neurons. The embedding dimension is $d_{model} = 512$ it needs to be projected to a bigger vocabulary size n, this operation make sure the softmax is computed on the whole vocabulary.

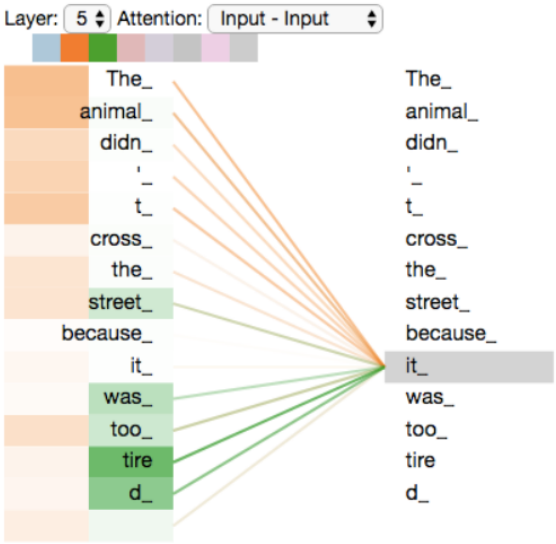


Figure 2.9: Two attention head are displayed relative to a i-th token the context of it is both associated with "the animal" and "tired".

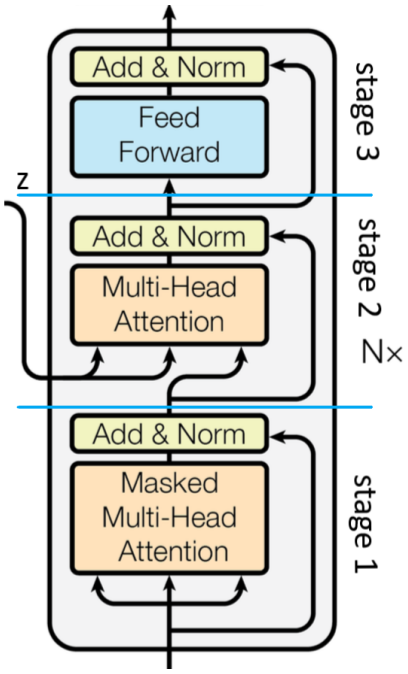


Figure 2.10: Decoder stages

Performance

The Transformer in Vaswani et al. [20] resulted better than any previous architecture in the translating task of English to German, based on the BLUE

score of 28. The training is highly parallelizable given the fact that operations are carried out in matrix form, only the decoding phase require different time stamps. In table 2.9 the computational complexity and maximum path length are compared, n is the sequence length, k is the size of kernel in convolutions, r is the size of the neighborhood for restricted self attention (attention with context window r) and d the representation dimensions (embedding size).

Layer Type	Complexity/Layer	Seq Op	Max Path Length
Self-Attention	$O\left(n^2 \cdot d\right)$	$O(1)$	$O(1)$
Recurrent	$O\left(n \cdot d^2\right)$	$O(n)$	$O(n)$
Convolutional	$O\left(k \cdot n \cdot d^2\right)$	$O(1)$	$O\left(\log_k(n)\right)$
Restricted-Attention	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 2.9: Computational complexity of self attention respect other models

Clearly for short sequence of symbols RNNs or LSTMs are still the best solution, but when n is larger than d Self attention is the winner, because it can access faster the long range dependencies.

2.4 NLP metrics

In the NMT world objective metrics are used to evaluate models, this quantitative metric is usually based on statistical analysis of words, one of the most important metric is BLUE (Bilingual Evaluation Understudy) proposed by Papineni et al. in 2002 [25].

A simple question arise when comparing translations, if a person don't understand the translated language how can the evaluation be possible without human judgment, one of the simplest way is to count the number of words present in the machine translated sentence over the human translated sentence with the hypothesis that we are in possess a good data-set of pairwise translations. The BLUE score is based on this principle but with other deeper consideration.

First of all it is more accurate to compare the machine translation with a reference translation, given that it is also possible to compare a ML model versus another ML model or simply there are many optimal reference translations. Once the machine translation is presented and the reference is also given, it is possible to measure metrics such as Precision, Recall, F1, BLUE and Rogue_L these are the main metrics used in this thesis.

Precision is the number of n-grams present in both the model and reference over the number of n-grams in the model.

$$\frac{count_{match}(n-gram)}{count_{model}(n-gram)} \tag{2.16}$$

In eq. (2.16) for n equals to length of sequential words.

Recall is the number of n-grams present in both the model and reference over the number of n-grams in the reference. In eq. (2.17) this measure is great for concise generative models where brevity is not considered as a negative impact.

$$\frac{count_{match}(n-gram)}{count_{ref}(n-gram)} \tag{2.17}$$

F1 The F1 is a combined formulation equal to 2 times precision times recall over sum of precision and recall.For eq. (2.18) the formulation can also have weights multiplied by precision and recall to adjust the importance of one feature over another.

$$2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.18}$$

Rogue or better Rougue_L is used as a special recall oriented score which measure the longest common sequence (LCS) over the whole sequence. In eq. (2.19), the Rougue_L takes into account the order of words and usually for n equal to 4 it is a good parameter for NMT task.

$$\frac{LCS(n-gram)}{count(n-gram)} \tag{2.19}$$

BLEU is one of the most popular metric it was created to ease the human emulators' work and accelerate the NMT R&D. The idea behind it is the same of the n-gram precision but with a brevity penalty, this penalty is a function in the interval [0,1] it assign lower marks for shorter generated translation respect to the reference one. The BLEU-n is usually used with n equal to 4. The Blue score has many variants and the variant used in the thesis is the smoothed Bleu score expressed in 0-100 scale.

$$\text{BLEU} = \min\left(1, \exp\left(1 - \frac{\text{ref-len}}{\text{out-len}}\right)\right)\left(\prod_{i=1}^{4} \text{precision}_i\right)^{1/4} \tag{2.20}$$

2.5 Objectives

The objective of SolidtySummarizer as stated in the introductions is to help the understanding and readability of smart contracts. Following the analysis of current research areas, SMTranslator is the first tool to summarize Solidity source code in natural language. It is a template-base method which lack in accuracy table 2.1 and flexibility of the generated descriptions. After careful analysis and study of the state of the art, if NMT is applied to Solidity it can greatly improve the code summarization task performed by SMTranslator breaking the boundaries imposed by template based generation of natural language. For this reason SolidtySummarizer exploits an existing Transformer based model called NeuralCodeSum.

Chapter 3

Comment generation

C++, Python and JavaScript heavily inspired the Solidity programming language[7]. The comment style is very similar to the last one: // double slash is used for single-line comment, /∗∗/ slash asterisk and asterisk slash are used respectively to open multi line comments and to close multi line comments. As described in section 2.3 pairs of code comment are used to train the encoder decoder model, and a big data set is needed in order to produce a valid model.

Unlike *NeuralCodeSum* project [21], where the data-set is obtained from previous works. Especially from (Barone, 2017) [26] for python data-set and (Hu, 2018) [27] for java data-set, there was not any code summarization study on solidity language. Therefore no pre-existing data-set is available, SmartContractDatabase `https://github.com/MatteoZhang/SmartContractDatabase` is a repository created for this thesis. It was built with different tools: the first one is represented by a web scraper created by fellow student Fabrizio Trovato, which downloads solidity contracts from Etherscan.io [28], the second part is a solidity parser tool which divides code and comments. There are many solidity parsers on GitHub, but the parsed output contains only the codes and comments are always neglected. Moreover, the parsed code does not highlight the functions which are essential to understand how the code works. The following sections describe in detail the parser logic.

3.1 Solidity Source Code

The source code are provided by Etherscan [28], using SmartContractDatabase is possible to obtain the data and process it. The formatted data is compatible with the Transformer model input.

The bottleneck of the system, described in fig. 3.1, is between Data collector and the Etherscan website , while the development bottleneck is inside the Data cleaning scripts where many design iterations are made.

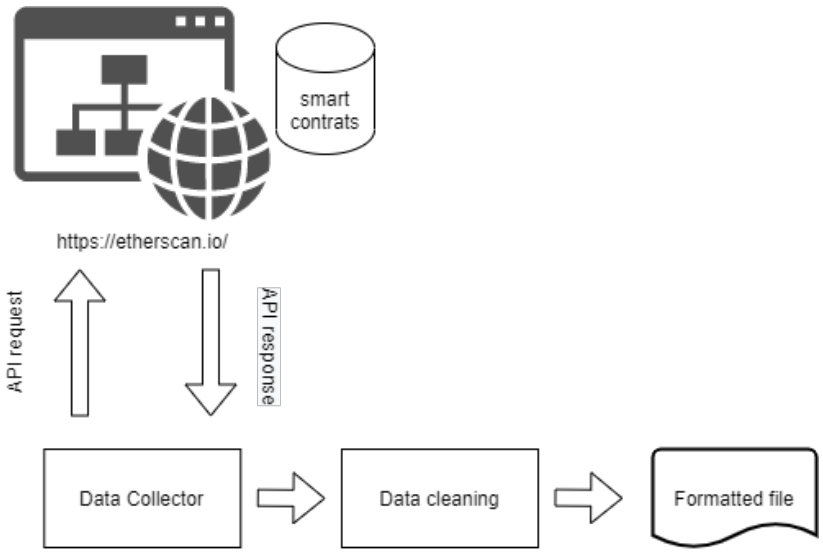


Figure 3.1: Architecture of SmartContractDatabase

3.1.1 Data collection

The data collection can be done with two types of tools one is the web scraper and the other one is using official APIs (Application Programming Interface) provided by Etherscan [28]. The web scraper is an automated tool that read the HTML file of a website and extracts relevant information exploiting the fact that most of the web pages are machine-generated. This approach is very slow, after some experimentation, 1 contract per 10 seconds is found to be the upper limit speed to download Solidity contracts, these automated and frequent requests to Etherscan.io can be seen as web attacks from the server side and then it will deny further requests. The easiest and fastest way to download smart contracts from Etherscan.io is to use the official APIs, which offer 5 API calls per second with the free plan.

An API call example is:

```
https://api.etherscan.io/api?module=contract&action=
getsourcecode&address=<ADDRESS>&apikey=<APIKEY>
```

The ADDRESS field represent the smart contract address and APIKEY is an auto-generated key which the user can create inside its own personal page. The API request's response have the following structure:

```
{
    status:""
    message:""
    result:{
        0:{
            SourceCode:""
        }
        ABI:{...}
        ContractName:""
        CompilerVersion:""
        ...
    }
}
```

Inside the response, the most important field for this project is the SourceCode field which is in string format, in the case of multiple library import the plain string contains a nested json file with multiple source code. Only the source code is stored and written on files, the nested source code is not used because the imported libraries heavily increase the number of codes with same functionality and the most common library is Safe-Math which limits the overflow of math operations.

The Etherscan website [28] provides also a CSV list of Verified contract addresses of which the code publishers have provided a corresponding Open Source license for re-distribution. This list contains the last 10000 smart contract address with verified code.

> *"Source code verification provides transparency for users interacting with smart contracts. By uploading the source code, Etherscan will match the compiled code with that on the blockchain. Just like contracts, a "smart contract" should provide end users with more information on what they are "digitally signing" for and give users an opportunity to audit the code to independently verify that it actually does what it is supposed to do.[28]"*

Although the smart contracts are "verified", displaying the source code doesn't mean it is secure but at least there is a degree of transparency.

Another way to download smart contracts is to use GitHub, on GitHub there are many repositories with solidity language, an advanced search can be done writing "*language* : *Solidity*" in the search bar, the repositories containing solidity language is around 1400, unlike Etherscan it is not possible to verify if the smart contracts are deployed or not and if it is used as smart

contracts or not. Downloading all 1400 repositories is not useful at all so only the most famous and used Solidity libraries are downloaded.

The total number of raw smart contracts collected from Etherscan and manually from GitHub are 13540, 500MB of raw data.

3.1.2 Data cleaning

The code summarization task: requires precise code comment pairs. A study on code summarization data-set highlighted that *"words in methods and comments tend to overlap, but in fact a vast majority of words are different (statistically 70%)"*[29]. This observation leads to a difficult problem because the words in comments represent high level concepts while source code represent low level implementation details. This problem is also known as *"concept assignment problem"* [30].

Identifying comments is an easy task, double slash for single line comment and slash asterisk for multiple line of comments. The position of the comments increase the magnitude of this problem: the comments can be above the code, inside the code and on the same line of the code, usually they are not under a code piece by convention.

```
1  // SPDX-License-Identifier: Apache-2.0
2
3  ////////////////////////////////////////////////////////////////////////////
4  //
5  //  Copyright (C) 2018-2021 Crossbar.io Technologies GmbH and contributors.
6  //
7  //  Licensed under the Apache License, Version 2.0 (the "License");
8  //  you may not use this file except in compliance with the License.
9  //  You may obtain a copy of the License at
10 //
11 //      http://www.apache.org/licenses/LICENSE-2.0
12 //
13 //  Unless required by applicable law or agreed to in writing, software
14 //  distributed under the License is distributed on an "AS IS" BASIS,
15 //  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
16 //  See the License for the specific language governing permissions and
17 //  limitations under the License.
18 //
19 ////////////////////////////////////////////////////////////////////////////
```

Figure 3.2: Decorations and Copyright notices.
Decoration can be found also between functions.

The division of the code comment pair can be realized in two ways, the first is to locate the comment, then find the corresponding code, and the

the latter consists first in finding the code key, then find the corresponding comments. The first method has an intrinsic flaw. The comments do not always explain codes, but they are also used for decoration, copyright notice, licence, annotation of bugs and TODO lists fig. 3.2, so the natural conclusion is to start identifying the code keywords needed to find corresponding comments first. In this thesis, the chosen keywords are function, event, and modifier because they are the most useful code segments: for example by analysing 100 pairs of code-comment, 95 lines of codes start with function, four start with an event, and one starts with modifier.

Code Separator

For each solidity document if the document contains solidity code, this code is pre-processed eliminating the empty lines and empty spaces before the first keyword of the code, then reading line by line all chosen keywords are located. Above the keyword, there is a convention to write multiple lines of codes to explain the function, nowadays semi automated comment generation tools are used to format the comment, most tool follows NatSpec Format (Ethereum Natural Language Specification)[7]. It generates tags like @notice, @dev, @param, @return to guide the developer during the comment generation.

The fig. 3.3 highlights the available options during the writing of the comments, this reduce in a certain way the "concept assignment problem". The other type of comments are mostly found inside the curly brackets of the function and in this case the comments usually don't follow precisely the NatSpec format as developers tend to comment the same line of the code to annotate, write between lines of code to remind bugs. This kind of comments are considered partially useful.

Tag		Context
`@title`	A title that should describe the contract/interface	contract, library, interface
`@author`	The name of the author	contract, library, interface
`@notice`	Explain to an end user what this does	contract, library, interface, function, public state variable, event
`@dev`	Explain to a developer any extra details	contract, library, interface, function, state variable, event
`@param`	Documents a parameter just like in Doxygen (must be followed by parameter name)	function, event
`@return`	Documents the return variables of a contract's function	function, public state variable
`@inheritdoc`	Copies all missing tags from the base function (must be followed by the contract name)	function, public state variable
`@custom:...`	Custom tag, semantics is application-defined	everywhere

Figure 3.3: Natspec Format comments tags, inspired by doxygen tags

An example of code showing the position of comments is:

```
1
2  // decorations = = = = = = =
3  /**
4      Natspec format
5      @notice this function ...
6  */
7  function FunctionName(){
8      /*
9          Multi line comments
10     */
11     //single line comments
12     loops and conditional statements
13     variables = ... // inline comments
14     return statements
15 }
16 // decorations = = = = = = =
```

The solidity compiler "solc" can generate user documentation and developer

documentation thanks to NatSpec tags, these documentation are not enough for code summarization because not all the developer use the suggested tags and using the generated documentation, the solidity code functions are partially lost. The Algorithm 1 pseudo code describes the code and comment separation proposed in this thesis: once a keyword is found, the separation of code and comment starts, searching the comments above the keyword and inside brackets of the function returns a list of comments and a flag indicating the presence of comments. If there are comments then the system store them and write on an output file the stored code and comments.

Algorithm 1: Code Comment Separator

```
input directory initialization;
for file in directory do
    open file;
    if file have solidity extension;
     then
        read file;
        initialize keywords;
        create list for each line of the file;
        Eliminate empty lines in list;
        Eliminate white left space of codes in list;
        for line in list do
            if line starts with keyword then
                search for comments above the function;
                search for comments inside the function;
                if comments are not present near keyword then
                    go next line;
                else
                    memorize code without the comments inside;
                    memorize the comments;
    else
        remove the file;
    Write the memorized code and comments;
```

Pre-process tokens

The codes and comments obtained from the code-comment separator are full of decorations, composed words (CammelCase and separated_words) and non English words. In order to have a good data-set it is mandatory to transform composed words into single words and then eliminate not useful symbols in the decorations, for the comment side all non ASCII character are filtered.

Filtering

In order to obtain different pairs of code and comment as already described in section 3.1.1 library functions with its own comment can undermine the machine learning model and over-fit the results, being an input for code summarization task the best way to filter duplicates is to store only piece of code with different comments, this way the uniqueness of comments are maintained, duplicates will not appear.

Summarizing:

- filtered cases: <same code, same comment>
- allowed cased: <same code, different comment>, <different code, different comment>, <different code, same comment>

3.2 NLG with NeuralCodeSum

NeuralCodeSum is a GitHub repository which implements the Transformer model described in section 2.3, the Solidity code obtained from data-cleaning and pre-processing is adapted to the input format of this NLG model. The repository doesn't contain data and pre-trained models so the first step is to make sure that this code is debugged and works fine with the programming environment set up for it.

In this thesis the Colab environment is used as it offer out of box Python environment with easy to configure libraries and Notebook parameters. There is a saying that most of Machine learning project spend 80% of time collecting and preparing the data, well for this project actually most of the time spent is on the training of the data. Colab parameters: GPU Nvidia K80 / T4, GPU memory 12GB/16GB, GPU Memory Clock 0.82GHz / 1.59GHz, Performance 4.1 TFLOPS / 8.1 TFLOPS.

In detail, the setup includes creating scripts to run the machine learning algorithm, setting up the python environment with Pytorch, and running the training scripts. During the training, a temporary folder is created called "tmp". Inside the folder, there are training checkpoint files that contain the trained embedding and all the parameters up to the last valid Epoch. This way, if any interruptions occurs, the training can resume. At the end of the training, a model file containing the best embedding is stored inside "tmp" folder. This is the output of our training. For any generation task, a script loads the model and translates word by word the input sequence.

Java data-set

The first step after the setup is trying to use the original data-set in Java [27] and run the scripts to create a java translator model "code2jdoc" (arbitrary name assigned as model filename with the embeddings), to train the model the only file to run is the bash file "transformer.sh" where the hyper-parameter are set and both input files, output directory are provided. The parameters are described in table 3.1.

The data-set parameters are in fig. 3.4. After 200 epochs of training the model "code2jdoc" is obtained. Using "code2jdoc" it is possible to generate comments from functions using the bash script "generate.sh", the fig. 3.5 taken from the NeuralCodeSum paper shows qualitatively the generated comments respect to the human written one, now taking the same code comment pair and using our "code2jdoc" to generate the comment, the predicted output

	Hyper-parameter	Value
Embedding	k	16
Model	l	6
	h	8
	d_{model}	512
	d_k, d_v	64
	d_{ff}	2048
Training	dropout	0.2
	optimizer	Adam
	learning rate	0.0001
	batch size	32
Testing	beam size	4

Table 3.1: Hyper parameter used to train the models

Attribute	Train	Valid	Test	Fullset
Records	69708	8714	8714	87136
Function Tokens	8371911	1042466	1055733	10470110
Javadoc Tokens	1235295	155876	153407	1544578
Unique Function Tokens	36202	15317	15131	66650
Unique Javadoc Tokens	28047	9555	9293	46895
Avg. Function Length	120.10	119.63	121.15	120.16
Avg. Javadoc Length	17.72	17.89	17.60	17.73

Figure 3.4: Java data-set statistics

is "evaluates the xpath expression as a single expression." which means that "code2jdoc" reproduced perfectly the results obtained in the reference paper.

Knowing the fact that the code works and it can potentially generate good quality comments in java, it is time to apply the model "code2jdoc" on solidity code instead, in order to see if pre-trained model on other programming language can produce same results or it is necessary to train another model on solidity data-set. On the test set of 13021 code-comment pairs, the generated comments doesn't perform well with respect to java comments, the obtained testing metrics are shown in table 3.2

The results of the model with respect of solidity code are not sufficient for the comment generation task, because the difference in raw data-set is too large and in order to generate better results, there is the need to train

```
public static String selectText(XPathExpression expr, Node context) {
    try {
        return (String)expr.evaluate(context, XPathConstants.STRING );
    } catch (XPathExpressionException e) {
        throw new XmlException(e);
    }
}
```

Base Model: evaluates the xpath expression to a xpath expression .
Full Model w/o Relative Position: evaluates the xpath expression .
Full Model w/o Copy Attention Attention: evaluates the xpath expression as a single element .
Full Model: evaluates the xpath expression as a text string .
Human Written: evaluates the xpath expression as text .

Figure 3.5: NeuralCodeSum generated code qualitative comparison between different parameters, code2jdoc is the Full model

Model	Blue	Rouge_L	Precision	Recall	F1
code2jdoc	43.48	53.86	59.19	57.04	55.64
code2jdoc on solidiy	5.42	10.68	23.09	10.51	12.24

Table 3.2: Code2jdoc model results
and application on solidity test dataset

from scratch another model on the solidity training data-set.

Solidity data-set

The original repository was adapted to generate solidity comments and the following changes are made: bash script for the training and testing and input constant files. This minor changes let any user to train and use the model for any other programming language. The data-set parameters are in fig. 3.6. The model trained is called "code2sol_XL" fig. 3.7, and in example

Attribute	Train	Valid	Test	Fullset
Records	60000	13021	13021	86042
Function Tokens	2320474	665144	774032	3759650
Javadoc Tokens	2654547	504859	502354	3661760
Unique Function Tokens	9045	10871	11735	31651
Unique Javadoc Tokens	10287	12051	12944	35282
Avg. Function Length	38.67	51.08	59.44	43.70
Avg. Javadoc Length	44.24	38.77	38.58	42.56

Figure 3.6: Solidity data-set statistics

below it is possible to see a qualitative example generated by "code2jdoc"

and "code2sol_XL" models:

```
function transfer Any ERC20Tokens(address token Addr, address
    to, uint amount) public only Owner {
    require((token Addr ! token Address) || (now>admin
   Claimable Time));
    Token(token Addr);
    transfer(to, amount);
}
```

The function is tokenized substituting CammelCase and snake_case with unique words as described in section 3.1.2.
The generated comments are:

- code2jdoc : "assigns an address from a string to a purely address and a kenlm any that is only included in an ip address." with score "bleu": 0.039, "rouge_l": 0.036
- code2sol_XL: "function to allow admin to claim other erc20 tokens sent to this contract (by mistake) with score "bleu": 0.41, "rouge_l": 0.69
- Human Written: function to allow admin to claim other erc20 tokens sent to this contract (by mistake) admin cannot transfer out yfox from this smart contract till 1 month after staking ends.

This example is very significant the average comment generated in "code2jdoc" is not trained to explain domain specific concepts instead "code2sol_XL" perform a better generation on average, which means the training of the latter model is successful. Other trained models with only one subset of train data are reported in section 3.2.

The fig. 3.7 shows that after 120 Epoch of training the model starts to converge and the Blue score rate of change starts to slow, moreover in some Epochs there are also negative rates.

Other models

Other attempts are made along the way. Before making the repository work as intended, smaller models were created to test different model parameters. Initial models were trained using different sizes of the data-set, for example, using only 1 thousand functions for training ("code2sol_M"), while "code2sol_L" uses 10 thousand as a training dataset. These two models over-fit during the training they both surpassing 50 as Blue score but when it is tested against test data, these models obtains low score near to 10 only.

Several hyper-parameters were also changed to verify the impact on the results. In particular, the changed hyper-parameters were the number of

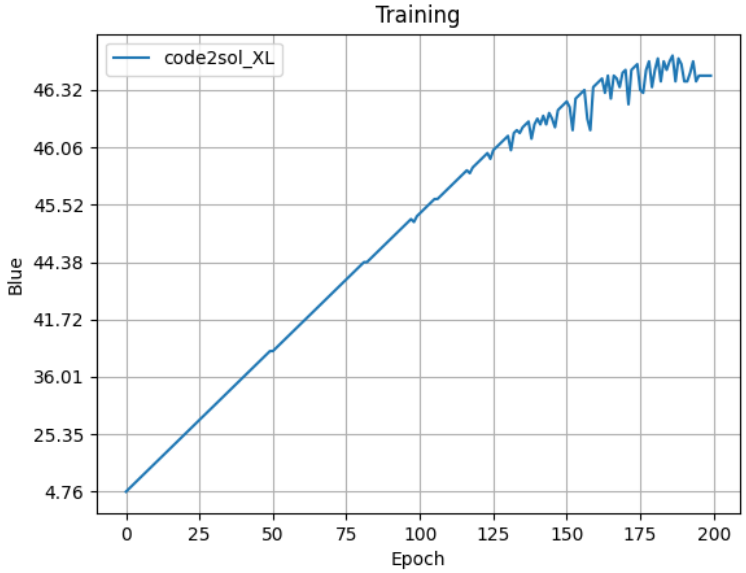


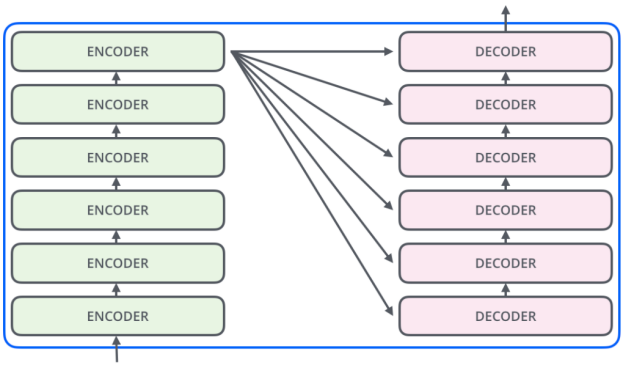
Figure 3.7: Training of the model, Blue score vs Epoch

layers N. They are stacked with 2, 4 and 8 pairs of identical encoders and decoders. The experiment results are the same as Varswani et al. [20], with more layers, the number of parameters to train grow and the overall score increase but not significantly around 0.5 to 1%. If there is no significant change, why the model is preferred with N equal to 6, which is the default value? The simple answer is that the perplexity is significantly lower. The perplexity is defined as the exponential average negative log-likelihood of a sequence eq. (3.1). With a sequence $X = x_0, x_1, ..., x_n$, the $\log p_\theta (x_i \mid x_{<i})$ is the likelihood of i-th conditioned on all preceding tokens $x_{<i}$. This metric is best indicated for auto-regressive causal language model (predictive models where the output depends linearly on the inputs).

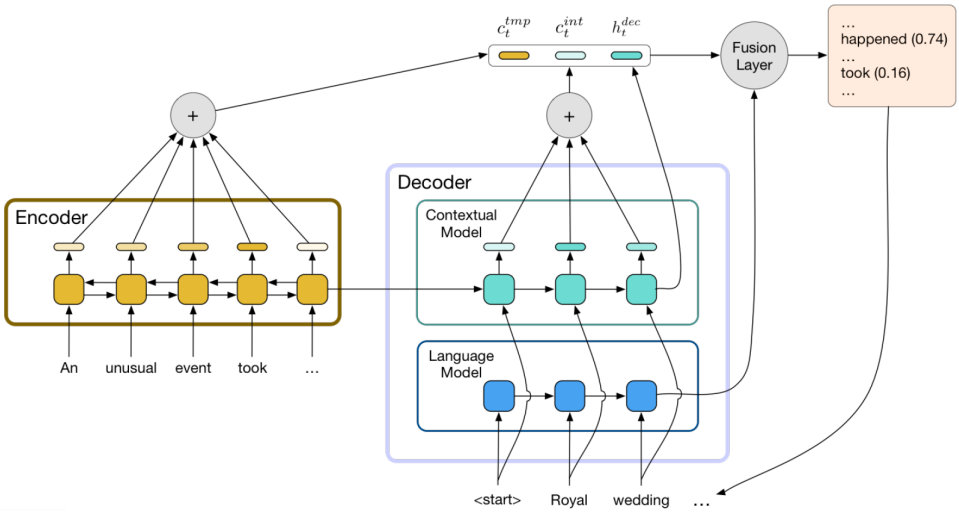
$$\mathrm{PPL}(X) = \exp\left\{-\frac{1}{t}\sum_{i}^{t}\log p_\theta\left(x_i \mid x_{<i}\right)\right\} \tag{3.1}$$

The last experiment is based on the suggestion of the NeuralCodeSum paper. It suggested the use of a different decoder called split decoder taken from (Kryscinski, 2018) [31]. The aim of Kryscinsky et al. to improve abstraction in text summarization, most of the summarization techniques uses extractive methods which are identifying important part of the text and create a summary while the abstractive method is used to capture human written summary style using paraphrases, technically with context words.

The encoder and decoder in a Transformer is described in fig. 3.8a, while the alternative implementation in fig. 3.8b. In the split decoder implementation, the decoder has two different outputs, one for the pair of (context, attention) and the other is the same output of the transformer decoder without split. The fusion layer computes the outputs and computes a final output embedding.



(a) Encoder Decoder in Transformer



(b) Network architecture with split decoder
where the context vectors and hidden states are fused to compute the output distribution.

Figure 3.8: Split Decoder implementation

3.2.1 NeuralCodeSum vs Vanilla Transformer

The original paper "Attention is all you need" [20], focused their work on Natural language to Natural language translation, while the NeuralCodeSum changed several things, especially on the input, which is code.

In source code many operations are commutative, the relation $a + b = b + a$ or the if conditions with $a > b = b < a$ are relative. This relative positional information can be added to the attention mechanism by setting a maximum action window and letting the vectors add the extra relative positional encoding. Defining self attention as eq. (3.2) where the simplified intermediate representation is z_i, Shaw et al.[32] proposed relative positioning changing the original eq. (3.3) and eq. (3.4) into eq. (3.5).

$$z_i = \sum_{j=1}^{n} \alpha_{ij} \left(x_j W^V\right) \tag{3.2}$$

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}} \tag{3.3}$$

$$e_{ij} = \frac{\left(x_i W^Q\right)\left(x_j W^K\right)^T}{\sqrt{d_z}} \tag{3.4}$$

$$\begin{aligned} z_i &= \sum_{j=1}^{n} \alpha_{ij} \left(x_j W^V + a_{ij}^V\right) \\ e_{ij} &= \frac{x_i W^Q \left(x_j W^K + a_{ij}^K\right)^T}{\sqrt{d_k}} \end{aligned} \tag{3.5}$$

The termsa_{ij}^V and a_{ij}^K are a relative positional representation of i and j, beyond a certain window Shaw et al. concludes that precise positional encoding is not useful. The eq. (3.6) shows the window k where 2k+1 relative positional representation are used. The used clip is in absolute terms, so the directional data is also ignored. In eq. (3.6), the term w^K is the positional representation to be added to the input embeddings.

$$\begin{gathered} a_{ij}^K = w_{\text{clip}(|j-i|,k)}^K, a_{ij}^V = w_{\text{clip}(|j-i|,k)}^V \\ \text{clip}(x,k) = \min(|x|,k) \end{gathered} \tag{3.6}$$

The programming language presents rare tokens, and thus copy attention must be implemented. Proposed by Nishida et al. [33] it could solve the problem by adding to the target Vocabulary rare tokens and making it the extended Vocabulary. The extended Vocabulary is essential for the name of

source code functions and technical words, which are not commonly used in the comment documentation.

Chapter 4

Evaluations

Evaluation is divided into two parts. In the first part, models produced by the fork of NeuralCodeSum are evaluated. The evaluation involves objective NLP metrics, explained in section 2.4. Then a survey with 7 participants is carried out. All of them have programming skills.

4.1 Quantitative and Qualitative analysis

From table 4.1, it is possible to observe the results of the trained models' performances on the test data set. The "code2jdoc" performs better than "code2sol_XL", but they share the same parameters and same architecture. From fig. 3.4 and fig. 3.6 it is possible to see the reason. On average, the documentation length of Solidity source code is far longer than the Java dataset: **38** words per sequence (Solidity) vs **17** words per sequence (Java). While the function length is 59 words vs 121 words, this means, during the training of "code2jdoc", the Transformer model have access to more inputs and context, which are more helpful during comment generation tasks. With shorter target sequences, in the Java dataset case, the Bleu score is computed with less penalty eq. (2.20). While for longer sequences, especially in the Solidity dataset, the prediction may lose few points.

Model	Blue	Rogue_L	Precision	Recall	F1
code2jdoc	**43.48**	**53.86**	**59.19**	**57.04**	**55.64**
code2sol_XL	32.15	45.34	59.01	47.07	48.32
code2sol_split	32.06	45.06	58.05	47.21	47.92

Table 4.1: Comparison between obtained metrics

In the Solidity dataset, the comments are not only a short summary of the

methods, but they also contain extra information such as long instructions to use a specific function, examples of usage and parameters. The testing results using the Solidity dataset is more robust than Java. With more records than the Java test set, it is acceptable to have a lower Blue score.

Model	Blue	Rogue_L
code2jdoc	43.48	53.86
NeuralCodeSum's model	44.58	54.76

Table 4.2: Comparison of the java code summarization between the model trained during the thesis and the original model made by Ahmad et al. [21]

The "code2sol_split" model does not improve the statistical metrics, but in table 4.1, it performed better for Recall with respect to "code2sol_XL", the original hypothesis of Kryscinsky et al. [31] is that the generated summary with split decoders would increase the Rogue score and surpass standard decoders but it is not in our case.

Another consideration is that there are fewer unique tokens in the whole Solidity dataset than the Java dataset, but this fact should only affect the training speed and not the prediction accuracy.

4.1.1 The Survey results

Finally, a survey was created to evaluate the Quality of the generated summaries. A total of 7 participants answered. The survey created for SoliditySummarizer is similar to the survey made by SMTranslator (Li, 2020) [17], the Q1 is substituted with "Q1.x Comment Analysis: I feel the comment is accurate and does not miss important information." because SolidtySummarizer has no user interface, which is why the average user is not intended to use the "tool" as it is. The Q6 is also substituted with "Are the generated comments useful for non-programmers?" as an ending question. In addition to the survey for the generated comments also a Survey with personal information is carried out to understand the background of the participants.

Both this thesis and SMTranslator used "Strongly disagree", "disagree", "agree" and "Strongly agree" in the answer options, but for SolidtySummarizer, "not sure" was added in order to individuate difficult methods to evaluate.

The Preliminary Questions are:

- What is your age?
- What is the highest degree or level of school you have completed?
- What is your programming level (any language)
- What is your experience with Ethereum Blockchain, Solidity and Smart Contracts?
- What is your main occupation?

The Evaluation Questions are:

- Q1.1-Q1.50 I feel the comment is accurate and does not miss important information.
- Q2 Accuracy: The explanations and summaries for a method are accurate.(The comments reflect the code)
- Q3 Readability: The generated summaries are easy to read and I can totally understand the meaning of each generated comments.
- Q4 Conciseness: The summaries do not contain unnecessary information.
- Q5 Instructiveness: I can easily use a specific method under the direction of the explanations.
- Q6 The generated comments are useful for non programmers.

The preliminary questions are made to understand the participants' backgrounds. Six out of Seven have a university degree, and one is a bachelor student. They all declare to be proficient or expert in one programming language, with the definition of proficient as "Good understanding of what the code is doing with occasional code references", and expert as "Can code with minimal references and understand the inner working of a language". 5 out of 7 have a good understanding of Blockchain concepts, and they are Beginner in Solidity, while the other one has advanced understanding of smart contracts and know-how to develop them (proficient in Solidity) and the last one knows only the Blockchain concepts and don't know how to program in Solidity. For the last preliminary question, there are in total 6 workers and 1 full-time student.

After the preliminary questions, the second part of the survey starts. The first question consists of 50 evaluations of the pairwise code comment snippets. This question is equivalent to Q6 of SMTranslator section 2.1, the result on Q1

is encouraging that more than 70% of methods are accurate for the comment generation task. Then on the general questions for Accuracy, Readability, Conciseness, Instructiveness and helpfulness for non-programmers are carried out (Q2-Q6), the results on Accuracy (Q2), Conciseness(Q4), Instructiveness(Q5) are overall positive while Readability(Q3) and Helpfulness(Q6) are polarized, the 2 participants rated Q3 as Strongly agree while 5 of them rated with Not Sure or Disagree, while for Q6, 4 participants rated Strongly agree and 3 Disagree. If compared to SMTranslator [17], the Readability aspect is a weakness of SolidditySummarizer. The generated comments are not polished and contain NatSpec tags with typical programmer's slang. The SMTranslator received better marks because templates are always readable and difficult to misinterpret. Under the Accuracy aspect, instead, SolidditySummarizer received 1 Strongly agree and 6 Agree marks while SMTranslator has 6 Agree and 4 Disagree, as already explained in chapter 2, templates not always are exhaustive and accurate, for too simple or too complex functions templates may lack flexibility. The results and the comparison of the surveys are presented in table 4.4.

Question	S agree	Agree	Not sure	Disagree	S disagree
Q1	38%	35.43%	18%	7.14%	1.43%
Q2	1	6/7	0	0	0
Q3	2/7	0	4/7	1/7	0
Q4	4/7	1/7	2/7	0	0
Q5	1/7	4/7	2/7	0	0
Q6	4/7	0	0	3/7	0

Table 4.3: Questions asked in the questionnaire, and participants opinions.

For the Q1 in table 4.3 if a linear value is assigned to the answer options: 5 for Strongly agree, 4 for Agree, 3 for Not sure, 2 for Disagree and 1 for Strongly Disagree. The medium mark is 4.01 with a small variance of 0.8, so in general, the participant agrees with the generated comments.

4.2 Code Comment Examples

The results obtained in chapter 4 can be compared with SMTranslator because the methods are taken from cited solidity smart contracts fig. 4.1 used in the authors' paper (Li, 2020) [17].

In Q1 (I feel the comment is accurate and does not miss important information.), Fifty code snippets are shown. Examples of code snippets,

Question	S agree	Agree	Not sure	Disagree	S disagree
Q1	38%	35.43%	18%	7.14%	1.43%
Q6*	0%	64%	0	23%	13%
Q2	1/7	6/7	0	0	0
Q2*	0	6/10	4/10	0	0
Q3	2/7	0	4/7	1/7	0
Q3*	9/10	1/10	0	0	0
Q4	4/7	1/7	2/7	0	0
Q4*	9/10	1/10	0	0	0
Q5	1/7	4/7	2/7	0	0
Q5*	0	3/4	1/4	0	0
Q6	4/7	0	0	3/7	0

Table 4.4: Comparison between the survey of SoliditySummarizer and SM-Translator (denoted with *).

divided by Strongly Agree to Strongly Disagree, are listed in this section. The chosen examples have most of one response type, i.e. Q1.7 have 5 Strongly Agree and 2 Agree marks, so it is very significant as Strongly Agree example.

Strongly Agree examples, these example are simple and basic, very easy to understand and self explanatory:
Q1.7

```
COMMENT: @ dev emitted when value tokens are moved from one
    account ( from ) to another ( to ) . note that value may
    be zero .
event Transfer(address indexed from, address indexed to,
    uint256 value);
```

Q1.9

```
COMMENT: @ dev get total supply of token
function totalSupply() external override view returns (
    uint256) {
     return _totalSupply;
}
```

Q1.18

```
COMMENT: @ dev returns the account approved for token id
    token . requirements : token id must exist .
function getApproved(uint256 _tokenId) public view returns (
    address _operator);
```

Type	Contract	Total Functions	Commented Functions	Size (KB)
Particial Description	MossCoin	13	5	7.8
	CCEToken	18	12	7.9
	DeusETH	26	3	6.8
	TutorialToken	19	12	7.7
	VocToken	26	9	7.3
	AMNToken	19	13	7.7
	ZmineToken	20	9	7.8
Vacuous Description	XBORNID	30	0	7.9
	XBR	32	0	7.8
	SiaCashCoin	29	0	7.1

Figure 4.1: All used smart contract for the survey questions

Q1.32

```
COMMENT: @ dev returns the greater of two numbers .

CODE: function max256(uint256 a, uint256 b) internal constant
    returns (uint256) { return a >= b ? a : b; }
```

Agree examples are, on average, more complex than other code comment pairs, and the generated comments carry more instructions or just information:
Q1.12

```
COMMENT: @ dev transfer tokens from one address to another .
         @ param from the address which you want to send
   tokens from
         @ param to the address which you want

function transferFrom( address _from, address _to, uint256
   _value ) external override returns (bool) {
        _transferByDefaultPartition(msg.sender, _from, _to,
   _value, ""); return true;
      }
```

Q1.14

```
COMMENT: @ dev internal function to add a token to this
   extension ' s ownership tracking data structures .
    @ param to address representing the new owner of the
   given token
```

```
3
4 function addTokenTo(address _to, uint256 _tokenId) internal {
5         require(tokenOwner[_tokenId] == address(0));
    tokenOwner[_tokenId] = _to;
6         ownedTokensCount[_to] = ownedTokensCount[_to].add(1);
7       }
```

Q1.15

```
1 COMMENT: @ dev tells whether an operator is approved by a
    given owner .
2          @ param owner owner address which you want to query
    the approval of @ param operator operator
3
4 function isApprovedForAll(address _owner, address _operator)
    public view returns (bool) {
5         return operatorApprovals[_owner][_operator];
6       }
```

Q1.22

```
1 COMMENT: @ dev extends transfer from method with event when
    the callback failed .
2          @ param from the address of the sender .
3          @ param to the address of the recipient
4
5 function transferFrom(address _from, address _to, uint256
    _value) public whenNotPaused returns (bool) {
6         return super.transferFrom(_from, _to, _value);
7       }
```

Q1.26

```
1 COMMENT: @ dev withdraw ether from contract @ param ether
    address uint256 amount of ether transfer
2
3 function withdraw() onlyOwner public { uint256 etherBalance =
     address(this).balance; owner.transfer(etherBalance); }
```

Q1.31

```
1 COMMENT: @ dev transfer tokens from one address to another
2          @ param from address the address which you want to
    send tokens from
3          @ param to address the address which you
4
5 function transferFrom(address _from, address _to, uint256
    _value) public returns (bool) {
6         require(_to != address(0));
```

```
        require(_value <= balances[_from]); require(_value <=
    allowed[_from][msg.sender]);
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.
    sender].sub(_value);
        emit Transfer(_from, _to, _value);
        return true;
      }
```

Not Sure examples are functions that also contains secondary functions, the generated comments are not intuitive at all, and for beginners, examples like Q1.4 may be challenging to understand. Other examples instead are confusing, like Q1.10 where the event is called Manager Registered. However, the generated comment is "emitted when a manager is changed":
Q1.4

```
COMMENT: @ dev increase the amount of tokens that an owner
    allowed to a spender.
         @ param partition name of the partition.
         @ param spender the address

function increaseAllowanceByPartition( bytes32 _partition,
    address _spender, uint256 _addedValue ) external returns (
    bool) {
        _approveByPartition( _partition, msg.sender, _spender
    , _allowedByPartition[_partition][msg.sender][_spender].
    add(_addedValue) );
        return true;
      }
```

Q1.8

```
COMMENT: @ dev get collateral occupied value .
         @ param partition name of the partition .

function _splitPartition(bytes32 _partition)internal pure
    returns (bytes4,bytes8,address){
        bytes4 prefix = bytes4(_partition);
        bytes8 subPartition = bytes8(_partition << 32);
        address addressPart = address(uint160(uint256(
    _partition)));
        return (prefix, subPartition, addressPart);
      }
```

Q1.10

```
COMMENT: @ notice emitted when a collateral manager is changed

event CollateralManagerRegistered(address collateralManager);
```

Q1.11

```
COMMENT: @ dev called by an { ierc777 } when need .

function tokensToTransfer( bytes4 functionSig, bytes32 partition, address operator, address from,
        address to, uint256 value, bytes calldata data, bytes calldata operatorData ) external;
```

Disagree examples are only a small portion, and the participants never marked unanimously one question with more than 3 Disagree marks, Q1.48 obtained 3 Disagree, 2 Not sure and 2 Agree, in the specific example, the keyword super may appear difficult for a beginner to use and they might misjudge the generated comment:
Q1.48

```
COMMENT: forward erc20 methods to upgraded contract if this one is deprecated

function transfer(address _to, uint256 _value) public whenNotPaused returns (bool) {
        return super.transfer(_to, _value);
    }
```

Strongly Disagree examples, are just short generated comments, the code2sol_XL model fail to generate useful information: Q39

```
COMMENT: transfer function

function multiTransfer(address[] _address, uint[] _value) public returns (bool) { for (uint i = 0; i < _address.length; i++) {
        transfer(_address[i], _value[i]); } return true;
    }
```

In Strongly Agree examples, usually, the functions are simple, and the comments are clear. These functions are the most used type of functions for smart contracts. They are very helpful for beginners. In Agree examples, the functions start to be longer with more parameters and instructions. These types are the majority, and the generated summary is pretty long and accurate. Not Sure examples are borderline useful to the reader because the generated

comments start to be shorter and not very descriptive of the functions. Finally, Disagree and Strongly Disagree examples are the minority. They are of two types: rare functions with wrong generated comments or very short comments that are not useful.

These pairs of code and comments are useful for beginners, but they may be difficult to understand for non-programmers. The gap is still large. An example can be the domain-specific language that Solidity use, the Ethereum is moving toward very domain-specific applications such as DeFis and NFTs the smart contracts will present two challenges for non-programmers: first to understand Solidity code within blockchain context and second domain-specific words like collateral, margin, swap, typical of Financial world.

Chapter 5

Conclusions

This thesis explored smart contract problems and state of the art Code Summarization techniques. It proposed a solution for helping the Solidity programmers to understand source code and annotating existing source code with comments. Two repositories are created along the way, SmartContract-Database and the fork of NeuralCodeSum, the first is able to collect and process Solidity source code, and the second is able to make generative models for comment creation.

The SolidditySummarizer effectively improved the summary created by SMTranslator, with state of the art techniques from Code Summarization models. Based on the comparison between the two surveys (table 4.3 and table 4.4), the Accuracy of the generated comments, by SolidditySummarizer, improved considerably from 75% to 65% with respect to SMTranslator. Also, from the results in the final survey and qualitative examples, it is possible to distinguish the improvements and weaknesses of SolidditySummarizer:

- Improvements:
  - The models produced by SolidditySummarizer are robust. Any piece of code can be translated. It does not depend on the length of the input.
  - Accuracy of the generated summary depends only on training data, while SMTranslator is limited by template-based models.
- Weakness:
  - The training data can limit the model performance, both in size and quality. High-quality data are difficult to find. In computer science, codes are usually reused, so the size of the training set is limited.

– The training time is long and if any major change is applied to the Solidity language, then a new training dataset must be created.

The comparison made in chapter 4 with SMTranslator underline that in general SolidditySummarizer is better, but with drawback like Readability of the generated comments as they are directly learned from source code, so the writing style is similar to programmers writing style, very concise and tagged with NatSpec style tags making it very schematic unlike sentences of templates.

For future development, source code generation may be a direction. Using these two repositories as basis or creating a user interface like a web app is another direction.

In conclusion, this thesis achieved its initial objective, which is to improve the current state of the art represented by Li et al. [17] by researching Machine Learning methodologies and models, in particular the Transformer architecture.

List of Figures

List of Tables

Bibliography

[1] Buterin Vitalik. Ethereum whitepaper. *URL: https://ethereum.org/it/whitepaper/*, 2013.

[2] Satoshi Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin.org/bitcoin.pdf*, 2008.

[3] Bitcoin wiki. Bitcoin script. *URL: https://en.bitcoin.it/wiki/Script*, accessed 2021.

[4] Benjamin Gramlich. Smart contract languages: A thorough comparison. *ResearchGate Preprint*, 2020.

[5] Tiobe index. https://www.tiobe.com/tiobe-index/, 2021. [Online; accessed 1/9/2021].

[6] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.

[7] Solidity documentation. https://docs.soliditylang.org/en/v0.8.6/index.html, 2021. [Online; accessed 1/11/2020].

[8] Bloomberg. https://www.bloomberg.com/features/2017-the-ether-thief/, 2017. [Online; accessed 1/9/2021].

[9] Hacking distributed. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 2016. [Online; accessed 1/9/2021].

[10] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1371–1385, 2018.

[11] Christopher K Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 210–215. IEEE, 2016.

[12] Olivia Choudhury, Nolan Rudolph, Issa Sylla, Noor Fairoza, and Amar Das. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 963–970. IEEE, 2018.

[13] Christopher D Clack, Vikram A Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 2016.

[14] Dianhui Mao, Fan Wang, Yalei Wang, and Zhihao Hao. Visual and user-defined smart contract designing system based on automatic coding. *Ieee Access*, 7:73131–73143, 2019.

[15] Luca Guida and Florian Daniel. Supporting reuse of smart contracts through service orientation and assisted development. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 59–68. IEEE, 2019.

[16] Emanuel Regnath and Sebastian Steinhorst. Smaconat: Smart contracts in natural language. In *2018 Forum on Specification & Design Languages (FDL)*, pages 5–16. IEEE, 2018.

[17] Ming Li, Anjia Yang, and Xinkai Chen. Towards interpreting smart contract against contract fraud: A practical and automatic realization. *IACR Cryptol. ePrint Arch.*, 2020:574, 2020.

[18] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.

[19] Yuxiang Zhu and Minxue Pan. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352*, 2019.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[21] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[23] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[26] Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*, 2017.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge.(2018). In *Proceedings of the 27th International Joint Conference on Artificial Intelli-gence (IJCAI 2018), Stockholm, Sweden, 2018 July 13*, volume 19, pages 2269–2275, 2018.

[28] Etherscan.io. https://etherscan.io/apis#contracts, 2021. [Online; accessed 1/12/2020].

[29] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. *arXiv preprint arXiv:1904.02660*, 2019.

[30] Ted J Biggerstaff, Bharat G Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *[1993] Proceedings Working Conference on Reverse Engineering*, pages 27–43. IEEE, 1993.

[31] Wojciech Kryściński, Romain Paulus, Caiming Xiong, and Richard Socher. Improving abstraction in text summarization. *arXiv preprint arXiv:1808.07913*, 2018.

[32] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

[33] Kyosuke Nishida, Itsumi Saito, Kosuke Nishida, Kazutoshi Shinoda, Atsushi Otsuka, Hisako Asano, and Junji Tomita. Multi-style generative reading comprehension. *arXiv preprint arXiv:1901.02262*, 2019.