Secure code synthesis

André Storhaug

Spring 2022

Master of Science in Computer Science
Supervisor: Jingyue Li - Associate Professor at NTNU

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

Abstract

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

Sammendrag

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

Acknowledgement

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

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

André Storhaug, Trondheim 28.05.2022

Contents

Figures

Tables

Code Listings

Acronyms

AST	Abstract Syntax Tree. 14
Att-BLSTM	Attention-Based Bidirectional Long Short-Term Memory. 14
AWD-LSTM	Average Stochastic Gradient Descent Weighted Dropped LSTM. 14
CNN	Convolutional Neural Network. 14
dBFT	delegated Byzantine Fault Tolerance. 6
DNN	Deep Neural Network. 14
DPoS	Delegated Proof of Stake. 6, 7
DT	Desision Tree. 14
GAN	Generative Adversarial Network. 14
GNN	Graph Neural Network. 14
IR	Intermediate Representation. ix, *Glossary:* Intermediate Representation
LPoS	Leased Proof of Stake. 7
LSTM	Long Short-Term Memory. 14
ML	Machine Learning. i, 1, 10, 11
NFT	Non Fungible Tokens. ix, 8, *Glossary:* Non Fungible Tokens
PBFT	Practical Byzantine Fault Tolerance. 6
PoA	Proof of Authority. 7
PoS	Proof of Stake. 6, 7
PoW	Proof of Work. 6

RF Random Forest. 14

SC Smart Contract. i, vi, viii, 2, 3, 8–10, 13, 17–20, 27, 28

SLR Systematic Literature Review. i, 28

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

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

Glossary

fork A blockchain thath diverges into two potential paths is called a fork. 5

Non Fungible Tokens A type of token that is unique. 8

Chapter 1

Introduction

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

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

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

In order to better secure automatic code generation, this thesis purposes a novel approach for use of ML models in large-scale transformer based code generation pipelines to ensure secure generated code. In order to demonstrate the approach, this thesis will focus on generating secure code for smart contracts (Solidity). Smart contracts have an exceptional high demand for security, as vulnerabilities can not be fixed after a contract is deployed. Due to most blockchains' monetary and anonymous nature, they pose as a desirable target for adversaries and manipulators [1]. Further, SCs tend to be rather short and simple, making a good fit for generated code. The research questions addressed in this thesis are:

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

The specific contributions of this thesis are as follows:

- Fine-tuned transformer-based language model for Smart Contract code generation.
- Novel secure code generation method.
- Identification of open issues, possible solutions to mitigate these issues, and future directions to advance the state of research in the domain.

The rest of this paper is organized as follows. Chapter 2 describes the background of the project. The research related to this document is commented in Chapter 3. **??** describes the methods used to implement the secure code generation. Chapter 7 describes the results of the project, and Chapter 8 discuss the findings. Identified future work is presented in Chapter 9. Chapter 10 presents final remarks and concludes the thesis.

Chapter 2

Background

This chapter introduces the necessary background information for this study. First, a brief introduction to blockchain technology is provided in Section 2.8 and then the concept of Smart Contracts (SCs) is introduced in Section 2.9. Finally, in Section 2.9.1, the most popular SC vulnerabilities are described.

2.1 Transfer learning

2.2 Natural language processing

2.2.1 Text preprocessing

2.2.2 Vector space model

Word2Vec

Term frequency-inverse document frequency

2.3 Machine Learning

2.3.1 Machine learning models

2.3.2 Linear models

Logistic regression

Support vector machines

Artificial neural networks

2.3.3 Deep learning

2.4 Cluster Analysis

2.5 Transformer

2.6 Performance metric

2.6.1 Precision

2.6.2 Recall

2.6.3 F1-score

2.6.4 Blue-score

2.7 String metric

2.7.1 Jaccard index

2.8 Blockchain

A blockchain is a growing list of records that are linked together by a cryptographic hash. Each record is called a block. The blocks contain a cryptographic hash of the previous block, a timestamp, and transactional data. By time-stamping a block, this proves that the transaction data existed when the block was published in order to get into its hash. Since all blocks contains the hash of the previous block, they end up forming a chain. In order to tamper with a block in the chain, this also requires altering all subsequent blocks. Blockchains are therefore resistant to modification. The longer the chain, the more secure it is.

Typically, blockchains are managed by a peer-to-peer network, resulting in a publicly distributed ledger. The network is composed of nodes that are connected to each other. The nodes collectively adhere to a protocol in order to communicate and validate new blocks. Blockchain records are possible to alter through a fork. However, blockchains can be considered secure by design and presents a distributed computing system with high Byzantine fault tolerance [2].

The blockchain technology was popularized by Bitcoin in 2008. Satoshi Nakamoto introduced the formal idea of blockchain as a peer-to-peer electronic cash system. It enabled users to conduct transactions without the need for a central authority. From Bitcoin sprang several other cryptocurrencies and blockchain platforms such as Ethereum, Litescoin, and Ripple. Table 2.1 shows an overview of the different blockchain platforms, including the different consensus protocols, programming languages, and execution environments used. It also shows the different types of blockchains, including public, private, and hybrid. If the platform also supplies a native currency (cryptocurrency), this is also shown.

Table 2.1: Comparison of blockchain platforms.

Refs.	Platform	Consensus	Runtime env.	Smart Contract Language	Type	Cryptocurrency[a]
[3]	Ethereum	PoW and PoS	Ethereum virtual machine (EVM)	Solidity	Public	Ether
[4]	Ethereum Classic	PoW	Ethereum virtual machine (EVM)	Solidity	Public	Ether
[5]	Bitcoin	PoW	Bitcoin script engine	Bitcoin script language	Public	Bitcoin
[6]	Hyperledger Fabric	PBFT	Docker	Go, Node.js, Java, Kotlin, Python	Permissioned	–
[7]	EOS	DPoS	WASM	C++	Public	EOS
[8]	NEO	dBFT	NEO virtual machine (NeoVM)	C#, Java, Go, Python	Public	NEO
[9]	Corda	PBFT	Deterministic JVM sandbox	CorDapp Design Language (CDL)	Permissioned	–
[10]	Tezos	PoS	Interprets Michelson	Michelson	Public	Tez (XTZ)
[11]	TRON	DPoS	TRON virtual machine (TVM)	TRON Solidity	Public	Tronix (TRX).
[12]	Æternity	Hybrid PoS PoW	Aeternity EVM (AEVM) and Fast æternity Transaction Engine (FATE)	Sophia	Public	Aeternity (AE)
[13]	RChain	PoS	Rho Virtual Machine	Rholang	Hybrid	REV

(*Continued on next page*)

Table 2.1: (*Continued*) Comparison of blockchain platforms.

Refs.	**Platform**	**Consensus**	**Runtime env.**	**Smart Contract Language**	**Type**	**Cryptocurrency**[a]
[14]	Cardano	PoS	Ethereum virtual machine EVM	Plutus (Functional)	Public	–
[15]	Aergo	DPoS	AERGO Virtual Machine (AVM)	Aergo Smart Contract Language (ASCL)	Hybrid	AERGO
[16]	QTUM	PoS	Qtum x86 virtual machine	Solidity	Public	QTUM
[17]	Waves	LPoS	Interprets RIDE	RIDE	Permissioned	Waves
[18]	Vechain	PoA	Custom EVM	Solidity	Hybrid	VeChain Token (VET)

[a] Name of the cryptocurrency. If none exists "–" is used.

2.8.1 Ethereum blockchain

2.9 Smart Contract

The term "Smart Contract" was introduced with the Ethereum platform in 2014. A Smart Contract (SC) is a program that is executed on a blockchain, enabling non-trusting parties to create an *agreement*. SCs have enabled several interesting new concepts, such as Non Fungible Tokens (NFT) and entirely new business models. Since Ethereum's introduction of SCs, the platform has kept its market share as the most popular SC blockchain platform. Ethereum is a open, decentralized platform that allows users to create, store, and transfer digital assets. Solidity is a programming language that is used to write smart contracts in Ethereum. Solidity is compiled down to bytecode, which is then deployed and stored on the blockchain. Ethereum also introduces the concept of gas. Ethereum describes gas as follows: “It is the fuel that allows it to operate, in the same way that a car needs gasoline to run.” [19]. The gas is used to pay for the cost of running the smart contract. This protects against malicious actors spamming the network [19]. The gas is paid in Wei, which is the smallest unit of Ethereum. Due to the immutable nature of blockchain technology, once a smart contract is deployed, it cannot be changed. This can have serious security implications, as vulnerable contracts can not be updated.

2.9.1 Security Vulnerabilities

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

Integer Overflow and Underflow

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

Transaction-Ordering Dependence

In blockchain systems, there is no guarantee on the execution order of transactions. A miner can influence the outcome of a transaction due to its own reordering

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

criteria. For example, a transaction that is dependent on another transaction to be executed first may not be executed. This can be exploited by malicious actors.

Broken Access Control

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

Code listing 2.1: Access control vulnerable Solidity Smart Contract code

```
contract SimpleSuicide {
    function sudicideAnyone() {
        selfdestruct(msg.sender);
    }
}
```

Timestamp Dependency

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

Code listing 2.2: Timestamp Dependency vulnerable Solidity Smart Contract code

```
1 contract Roulette {
2     uint public prevBlockTime; // One bet per block
3     constructor() external payable {} // Initially fund contract
4
5     // Fallback function used to make a bet
6     function () external payable {
7         require(msg.value == 5 ether); // Require 5 ether to play
```

```
        require(now != prevBlockTime); // Only 1 transaction per block
        prevBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.transfer(this.balance);
        }
    }
}
```

Reentrancy

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

Code listing 2.3: Reentrancy vulnerable Solidity Smart Contract code

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

2.10 Vulnerability detection

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

2.10.1 Symbolic execution

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

syntax analysis

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

Abstract interpretation

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

Data flow analysis

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

Fuzzy testing

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

Machine learning

Machine Learning (ML) is the study of computer algorithms that can automatically improve through the use of data. ML algorithms create a model based on training data. This model is then used to predict the outcome of new unseen data without being explicitly programmed to do so. Machine learning is a powerful tool for detecting vulnerabilities, as the following section shows.

Chapter 3

Related work

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

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

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

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

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

3.1 Code synthesis

3.2 Code comment analysis

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

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

3.3 OLD

This chapter provides a short overview of state-of-the-art secondary studies related to Smart Contract (SC) vulnerability analysis and detection. This is various literature that presents a survey or literature review nature.

Table 3.1: Existing ML-based smart contract vulnerability detection tools.

Refs.	Year	Name.[a]	Method	Feature engineering	Capability	Input
[20]	2018	Color-inspried	CNN	RGB image	Multi-label	EVM Bytecode
[21]	2019	Sequential learning	LSTM	Opcode embedding	Binary decision	EVM Opcode
[22]	2019	–	AST	AST statistics	Multi binary decision	Solidity
[23]	2020	NLP-inspried	AWD-LSTM	Opcode embedding	Multi-class	EVM Opcode
[24]	2020	Graph NN-based	GNN	Normalized Graph	Multi-class	Solidity
[25]	2020	Slicing matrix	CNN, RF	# opcode occurrence	Multi-class	EVM bytecode
[26]	2020	–	Att-BLSTM	Contract snippet, word embedding	Reentrancy	Solidity
[27]	2021	–	AST	# common child nodes	Multi-class	Solidity
[28]	2021	VSCL	DNN	CFG, N-gram model	Binary decision	EVM bytecode
[29]	2021	–	GAN	Code embedding	Binary decision	EVM bytecode
[30]	2021	ESCORT	LSTM, transfer learning	Word embedding	Multi-label	EVM Bytecode
[31]	2021	ContractWard	DT	Bigram model	Binary decision	EVM Opcode

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

3.4 Program synthesis

3.5 Code comment analysis

Chapter 4

Method

This chapter presents the research method used in this thesis. Firstly, the research motivation is presented, followed by the research questions defined for this thesis. Finally, the research method and design is explained.

4.1 Research Motivation

Existing solutions are slow. A total of X new contrats are deployed each day on the Ethereum network. Witth tthe speedup of the upgraded ethereum network, this is likely to result in an utterly spike in deployed contracts. Existing solutions are scarse. Further, the few ones available are based on symbolic analysis. Tthis is a very slow approach. It is therefore not possible to process all new contractts due to time resttrictions. An natural approach for combatting this is by leveraging deep learning techniques.

4.2 Research Questions

The research questions addressed in this thesis are:

RQ1. Officia magna irure ut occaecat cupidatat non sunt sunt.

4.3 Research Method and Design

4.4 Tools and Libraries

4.5 Hardware resources

4.5.1 IDUN High Performance Computing Platform

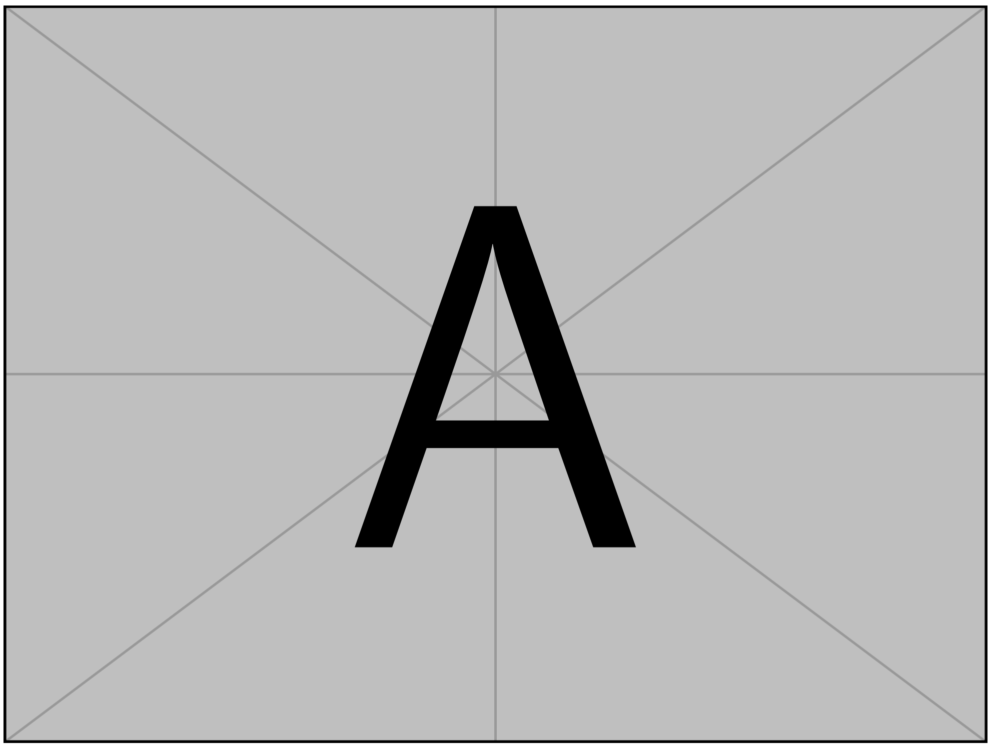

Figure 4.1: Flowchart of.....

Chapter 5

Data

This chapter introduces the necessary background information for this study. First, a brief introduction to blockchain technology is provided in Section 2.8 and then the concept of Smart Contracts (SCs) is introduced in Section 2.9. Finally, in Section 2.9.1, the most popular SC vulnerabilities are described.

5.1 Smart contract downloader

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

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

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

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

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

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

5.2 Datasets

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

5.2.1 Verified Smart Contracts

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

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

The current font size is: 6pt
The current font size is: 9pt
The current font size is: 10pt
The current font size is: 10.95pt
The current font size is: 12pt
1,0.5,0.5
0.7,1,0.7
0.7,0.7,1
5.11911in
Subsets:

Raw

Flattened

Inflated

Parsed

5.2.2 Verified Smart Contracts Audit

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

Subsets:

SoliDetector

Slither

5.2.3 Smart Contract Comments

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

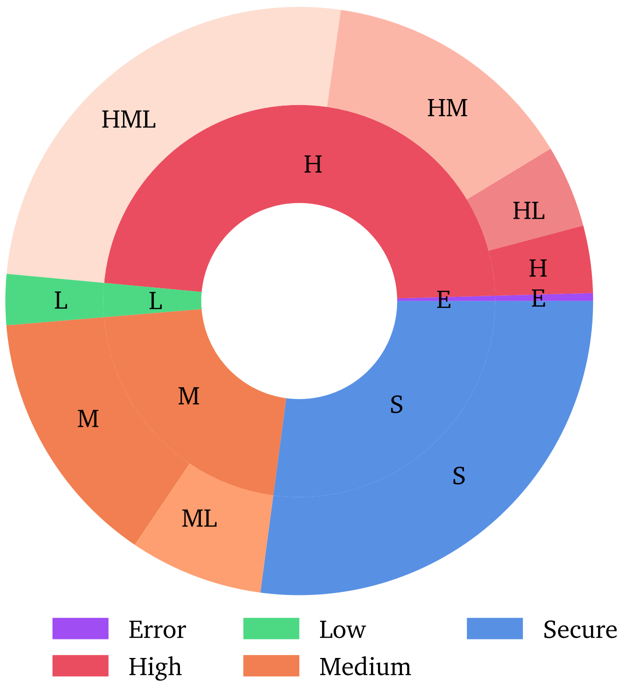

Figure 5.1: Doughnut chart over security levels, where each level occurs at least once in the SC. The inner ring shows the distribution of the occurrences of each level. The outer ring shows the additional security levels for each contract. For example, "HML" means that the contract has at least three vulnerabilities with the corresponding "High", "Medium", and "Low" security levels.

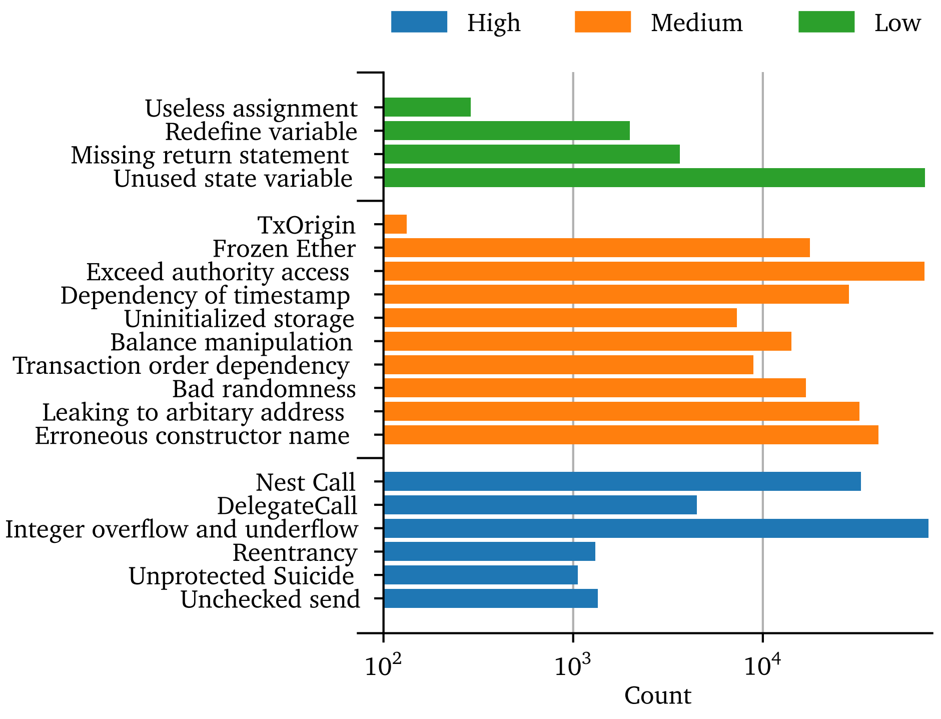

Figure 5.2: Histogram x of verified SCs on Ethereum

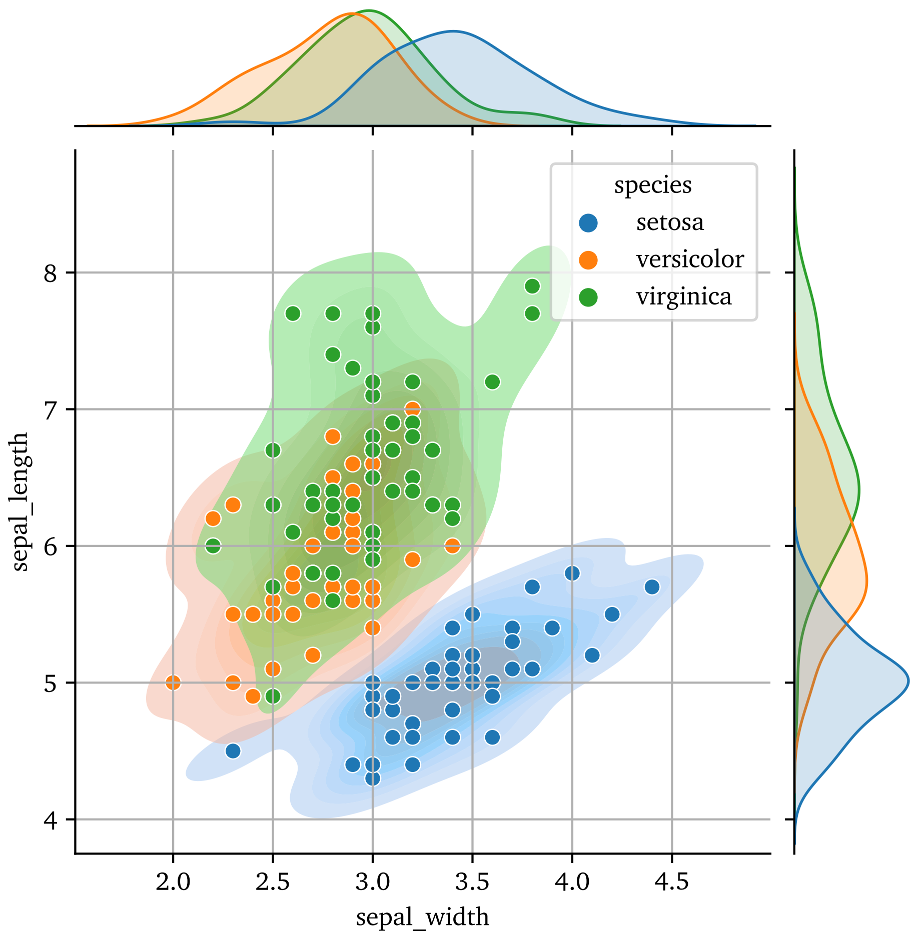

Figure 5.3: lol

Chapter 6

Architecture

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

Why did I select this model?

6.0.1 GPT-J

Table 6.1: Hyper parameters for GPT-J model

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

Table 6.3: Inclusion and exclusion criteria.

Hyper parameter	
stage	2
contiguous_gradients	true
reduce_scatter	true
reduce_bucket_size	2.000000e+08
allgather_partitions	true
allgather_bucket_size	2.000000e+08
overlap_comm	true
load_from_fp32_weights	true
elastic_checkpoint	false
offload_param	null
offload_optimizer	device: null
	nvme_path: null
	buffer_count: 4
	pin_memory: false
	pipeline_read: false
	pipeline_write: false
	fast_init: false
sub_group_size	1.000000e+09
prefetch_bucket_size	5.000000e+07
param_persistence_threshold	1.000000e+05
max_live_parameters	1.000000e+09
max_reuse_distance	1.000000e+09
gather_16bit_weights_on_model_save	false
ignore_unused_parameters	true
round_robin_gradients	false
legacy_stage1	false

Chapter 7

Research results

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

Chapter 8

Discussion

In this thesis, ...

8.1 Comparison with related work

Compared to related ...

8.2 Threats to Validity

Chapter 9

Future work

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

9.1 Comparison with related work

9.2 Threats to validity

Chapter 10

Conclusion

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

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

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

Bibliography

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

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

[3] Ethereum. “Ethereum.” (Jan. 2022), [Online]. Available: `https://ethereum.org/en/` (visited on 01/10/2022).

[4] Ethereum. “Ethereum classic.” (Dec. 2021), [Online]. Available: `https://ethereumclassic.org` (visited on 01/10/2022).

[5] B. Project. “Bitcoin.” (Jan. 2022), [Online]. Available: `https://bitcoin.org/en/` (visited on 01/10/2022).

[6] H. Foundation. “Hyperledger fabric.” (), [Online]. Available: `https://www.hyperledger.org/use/fabric` (visited on 10/27/2021).

[7] Block.one. “Eosio.” (Jan. 2022), [Online]. Available: `https://eos.io` (visited on 01/10/2022).

[8] N. Team. “Neo.” (Jan. 2022), [Online]. Available: `https://neo.org` (visited on 01/10/2022).

[9] r3. “Corda.” (Jan. 2022), [Online]. Available: `https://www.corda.net` (visited on 01/10/2022).

[10] tezoss. “Tezos.” (Jan. 2022), [Online]. Available: `https://www.tezos.com` (visited on 01/10/2022).

[11] tron dao. “Tron.” (Jan. 2022), [Online]. Available: `https://tron.network` (visited on 01/10/2022).

[12] æternity. “Æternity.” (Jan. 2022), [Online]. Available: `https://aeternity.com` (visited on 01/11/2022).

[13] rchain. “Rchain.” (Jan. 2021), [Online]. Available: `https://rchain.coop` (visited on 01/11/2022).

[14] Cardano. “Cardano.” (Jan. 2021), [Online]. Available: `https://cardano.org` (visited on 01/11/2022).

[15] Aergo. “Aergo.” (Jan. 2021), [Online]. Available: https://www.aergo.io (visited on 01/11/2022).

[16] Q. C. FOUNDATION. “Qtum.” (Jan. 2021), [Online]. Available: https://www.aergo.io (visited on 01/11/2022).

[17] Waves. “Wawes.” (Jan. 2021), [Online]. Available: https://waves.tech (visited on 01/11/2022).

[18] V. Foundation. “Vechain.” (Jan. 2021), [Online]. Available: https://www.vechain.org (visited on 01/11/2022).

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

[20] T. H.-D. Huang, *Hunting the ethereum smart contract: Color-inspired inspection of potential attacks*, 2018. arXiv: 1807.01868 [cs.CR].

[21] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, *Towards safer smart contracts: A sequence learning approach to detecting security threats*, 2019. arXiv: 1811.06632 [cs.CR].

[22] P. Momeni, Y. Wang, and R. Samavi, “Machine Learning Model for Smart Contracts Security Analysis,” in *2019 17TH INTERNATIONAL CONFERENCE ON PRIVACY, SECURITY AND TRUST (PST)*, Ghorbani, A and Ray, I and Lashkari, AH and Zhang, J and Lu, R, Ed., ser. Annual Conference on Privacy Security and Trust-PST, 17th International Conference on Privacy, Security and Trust (PST), Fredericton, CANADA, AUG 26-28, 2019, IEEE; Atlantic Canada Opportunities Agcy; TD Bank; IEEE New Brunswick Sect; CyberNB; Ignite Fredericton; ARMIS, 2019, 272–277, ISBN: 978-1-7281-3265-5.

[23] A. K. Gogineni, S. Swayamjyoti, D. Sahoo, K. K. Sahu, and R. kishore, *Multi-class classification of vulnerabilities in smart contracts using awd-lstm, with pre-trained encoder inspired from natural language processing*, 2020. arXiv: 2004.00362 [cs.IR].

[24] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, “Smart contract vulnerability detection using graph neural network,” in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, C. Bessiere, Ed., Main track, International Joint Conferences on Artificial Intelligence Organization, Jul. 2020, pp. 3283–3290. DOI: 10.24963/ijcai.2020/454. [Online]. Available: https://doi.org/10.24963/ijcai.2020/454.

[25] C. Xing, Z. Chen, L. Chen, X. Guo, Z. Zheng, and J. Li, “A new scheme of vulnerability analysis in smart contract with machine learning,” *WIRELESS NETWORKS*, 2020, ISSN: 1022-0038. DOI: {10.1007/s11276-020-02379-z}.

[26] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models," *IEEE ACCESS*, vol. 8, 19685–19695, 2020, ISSN: 2169-3536. DOI: {10.1109/ACCESS.2020.2969429}.

[27] Y. Xu, G. Hu, L. You, and C. Cao, "A Novel Machine Learning-Based Analysis Model for Smart Contract Vulnerability," *SECURITY AND COMMUNICATION NETWORKS*, vol. 2021, Aug. 2021, ISSN: 1939-0114. DOI: {10.1155/2021/5798033}.

[28] F. Mi, Z. Wang, C. Zhao, J. Guo, F. Ahmed, and L. Khan, "VSCL: Automating Vulnerability Detection in Smart Contracts with Deep Learning," in *2021 IEEE INTERNATIONAL CONFERENCE ON BLOCKCHAIN AND CRYPTOCURRENCY (ICBC)*, 3rd IEEE International Conference on Blockchain and Cryptocurrency (IEEE ICBC), ELECTR NETWORK, MAY 03-06, 2021, IEEE; IEEE Commun Soc; IBM; CSIRO, Data61, 2021, ISBN: 978-1-6654-3578-9. DOI: {10.1109/ICBC51069.2021.9461050}.

[29] H. Zhao, P. Su, Y. Wei, K. Gai, and M. Qiu, "GAN-Enabled Code Embedding for Reentrant Vulnerabilities Detection," in *KNOWLEDGE SCIENCE, ENGINEERING AND MANAGEMENT, PT III*, Qiu, H and Zhang, C and Fei, Z and Qiu, M and Kung, SY, Ed., ser. Lecture Notes in Artificial Intelligence, 14th International Conference on Knowledge Science, Engineering, and Management (KSEM), Tokyo, JAPAN, AUG 14-16, 2021, Springer LNCS; Waseda Univ; N Amer Chinese Talents Assoc; Longxiang High Tech Grp Inc, vol. 12817, 2021, 585–597, ISBN: 978-3-030-82153-1. DOI: {10.1007/978-3-030-82153-1_48}.

[30] O. Lutz, H. Chen, H. Fereidooni, C. Sendner, A. Dmitrienko, A. R. Sadeghi, and F. Koushanfar, *Escort: Ethereum smart contracts vulnerability detection using deep neural network and transfer learning*, 2021. arXiv: 2103.12607 [cs.CR].

[31] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2021. DOI: 10.1109/TNSE.2020.2968505.