Introduction

The EVM ecosystem faces two big problems which are caused by there being too many contracts and not enough auditors. Firstly, too many contracts are attempting to get audited, meaning they wait longer or apply for auditing from a worse firm. Secondly, there are too many contracts in the networks that exist out there for an auditor to account for the myriad of ways that it might be interacted with. The ecosystem needs a better tool that ensures a higher level of quality. In this paper the current issues facing the ecosystem and a proposed solution to help tackle this issue will be discussed.

## Literature Review

### Chess AI

“Deep Blue was intelligent the way your programmable alarm clock is intelligent. Not that losing to a $10 million alarm clock made me feel any better.” ― Garry Kasparov (1). The paper Deep Blue, written by Murray Campbell, A. Joseph Hoane Jr., and Feng-hsiung Hsu, overviews a chess-playing AI developed by IBM, which famously defeated the world champion Garry Kasparov in a six-game match in 1997. The system used custom hardware, allowing it to evaluate up to 2.5 million positions per second. The system relied on a heuristic evaluation function, which assigned values to different board positions based on factors such as the material balance, pawn structure, and king safety. Deep Blue's algorithms also incorporated a database of grandmaster-level games, which allowed the system to recognize common patterns and strategic motifs. By combining these techniques with a brute-force search of the game tree, Deep Blue was able to analyse many moves ahead and make highly sophisticated decisions. While Deep Blue's victory over Kasparov was a major milestone in the history of AI, the system was highly specialized and could not be easily adapted to other domains.

### Smart contracts

A smart contract (Dr Gavin Wood Ethereum 2022) is a self-executing program that automatically enforces the terms of an agreement between two or more parties. It is built on blockchain technology and allows for secure and transparent transactions without the need for intermediaries. Smart contracts are used in a wide range of applications, from financial services and real estate to supply chain management and voting systems. They can execute automatically and autonomously once the predefined conditions are met, making them a powerful tool for automating processes and reducing costs. Smart contracts have the potential to revolutionize many industries by providing a secure, efficient, and transparent way to conduct business transactions.

#### EVM

This links into the behaviour of the Ethereum Virtual Machine (EVM)(ethereum.org), the EVM is not like most virtual machines where if you had access to it, you could change anything that you want. The EVM is a system of state transition, given a state S and a transaction TX we can represent the state change as such:

The reason this distinction is important is because no failing transaction can be forced into the blockchain Crosby et al., (2015). The Ethereum Virtual Machine (EVM) is a software environment that allows developers to write and deploy smart contracts on the Ethereum blockchain. It is a powerful, decentralized computing platform that enables the execution of code in a secure and deterministic manner. The EVM uses a bytecode language that is designed to be executed on the virtual machine, allowing developers to write programs that are independent of any specific operating system or hardware. It also has a sophisticated gas system that helps to prevent malicious attacks by ensuring that executing code consumes a finite and limited number of resources. The EVM is a critical component of the Ethereum ecosystem, enabling developers to build decentralized applications and services that can interact with each other in a trust less and transparent way.

#### Transactions

[Transactions](https://ethereum.org/en/developers/docs/transactions/) on the Ethereum blockchain are the means by which value is transferred between addresses. Each transaction contains information such as the sender and recipient addresses, the amount of Ether being transferred, and an optional data field. Transactions are broadcast to the network, verified by nodes through a consensus mechanism, and added to the blockchain as part of a new block once validated. To execute a transaction, the sender must have a sufficient balance of Ether and must pay a fee in the form of "gas" to cover the cost of computation and storage on the network. Transactions on Ethereum are irreversible once they have been confirmed and added to the blockchain, making it a secure and reliable way to transfer value. In addition to simple Ether transfers, transactions can also be used to execute smart contracts and trigger automated actions on the blockchain.

### Exploits

#### What is an exploit?

The definition of an exploit is a difficult subject, there are two schools of thought, one is about intention and the second is “if the transaction doesn’t revert, I’ve done nothing wrong.” The first one states that an exploit can occur when a system is not used in the way that was intended. The second looks at the whole network that is being run, the contracts on it and the ways that you can interact with it and says that if they can do something they should be allowed to do it. This behaviour can be observed from the original DAO exploiter, Gazi Güçlütürk (2018): “I am disappointed by those who are characterizing the use of this intentional feature as "theft". I am making use of this explicitly coded feature as per the smart contract terms” Anonymous, (2022).

#### Network Exploit

A network exploit in this context is something that sits outside of the EVM that this paper focuses on, for example the [Bitcoin inflation bug](https://news.bitcoin.com/bitcoin-history-part-10-the-184-billion-btc-bug/) or the [Shanghai DOS attacks](https://www.infoq.com/news/2016/09/Ethereum-DOS-Attack/). These are things that even though are exploits, they are a deviation of the expected behaviour of the network which is different from a contract that runs on it. This is not an exploit of a contract and thus exploits of contracts that use an exploit in the network's implementation are not counted.

#### Singe contract

A single contract exploit is where only one contract is involved in the exploit (the exploited one) for example here is a list of common exploits <https://github.com/crytic/slither/wiki/Detector-Documentation> - slither, most of these are problems that can be exploited in a single transaction, not even needing to write a contract to perform the exploit. These can be things such as not using a NONCE when verifying signatures, or even more simply allowing the override of the “from” field instead of using “msg.sender” for an ERC-20 contract (see later). All these are things that can be easily checked for automatically and thus do not require the planned tool.

#### Multi contract

Chart

Description automatically generatedMulti contract exploits are more complex inherently as they have more moving parts, for example the [Harvest finance exploit in 2020](https://medium.com/harvest-finance/harvest-flashloan-economic-attack-post-mortem-3cf900d65217). In the graphic on the [left](https://twitter.com/bneiluj/status/1320686478486347778/photo/1) a tool was used to visualise all the transactions and calls made to all of the involved contract. This visualisation shows the complexity of the exploit, which would be exceedingly difficult to find manually. When you audit the code before deploying it you are assuming that your team is smarter than the rest of the world and will find everything. There are thousands of people searching exploits every day, they can check through a lot more things that your small team.

#### Flash loans

[Flash loans](https://eips.ethereum.org/EIPS/eip-3156) first introduced in 2019 by [Marble](https://medium.com/marbleorg/introducing-marble-a-smart-contract-bank-c9c438a12890) allow for the instant zero risk access to Billions of dollars of funds without any collateral. It works because it is enforced by code, a call can be passed and the amount being asked for to the lending contract which sends the funds to the requester, then makes the call that it was given and finally asks for the funds back with a small amount of interest. This all done within a single transaction, the funds are never “left” in the borrower’s account. Although research is being done on multi block flash loans which will provide even more opportunities to exploit contracts in the future.

#### MEV

### [MEV](https://ethereum.org/en/developers/docs/mev/#top) (Maximal Extractable Value) on Ethereum refers to the amount of value that can be extracted by miners or validators through various activities such as reordering, censorship, and transaction insertion. This value can come from sources such as transaction fees, arbitrage opportunities, and liquidations. MEV can have both positive and negative effects on the Ethereum ecosystem, as it can incentivize miners to act in ways that maximize their profits, but also create instability and inequality. To mitigate these effects, Ethereum developers are working on solutions such as MEV-Geth and FlashBots that aim to make MEV extraction more transparent and equitable.

### Current Tools and their methodology

#### Slither - Static analysis

[Slither](https://github.com/crytic/slither) is a tool that uses static analysis to identify security vulnerabilities in Solidity smart contracts written for the EVM. The tool employs a combination of both data flow and control flow analysis techniques to perform an examination of the contract's code. It first parses the source code into an abstract syntax tree (AST), then applies a variety of analysis techniques to detect potential vulnerabilities such as re-entrancy attacks, integer overflow/underflow, and use of uninitialized storage. Slither provides detailed feedback to the developer about the identified vulnerabilities, along with suggestions for remediation. It also includes a number of plugins that can be used to extend its functionality.

The use of static analysis is a valuable approach for identifying security vulnerabilities in smart contracts. Tools like Slither can help developers find and address security issues early in the development process before the contract is deployed on the blockchain. However, there are some limitations to this approach. For example, static analysis tools may not be able to detect all types of vulnerabilities and may produce false positives or false negatives. Additionally, the complexity of smart contracts and the lack of formal specifications can make it difficult to ensure complete coverage of all potential vulnerabilities. Despite these challenges, the use of static analysis tools like Slither represents a key step towards improving the security of smart contracts and ensuring the integrity of any contract.

#### Unit tests

##### Foundry

[Foundry](https://book.getfoundry.sh/) is a testing tool designed to aid developers to write comprehensive and efficient unit tests for EVM smart contracts. The tool enables developers to easily create test cases that cover a wide range of scenarios and edge cases. This allows them to identify and address potential vulnerabilities before deploying their contracts to the blockchain. Foundry provides a range of features to support unit testing, including support for a variety of test frameworks, automatic contract deployment, and the ability to simulate real-world conditions such as network latency and gas prices. Unit testing allows developers to test a large variety of features of a contract, this is done by simulating the smart contracts behaviour in a verity of potential environments. It also allows the testing of the contract for multiple network deployments with little reconfiguration being required.

##### Brownie – Unit tests

[Brownie](https://eth-brownie.readthedocs.io/en/stable/) is a smart contract testing tool for Ethereum Virtual Machine (EVM) development that is written in Python. It supports the testing of smart contracts written in Solidity and Vyper providing various functionalities, including automated contract testing, interactive console, and coverage reports. Despite its utility, Brownie has certain limitations which must be acknowledged. For instance, it is not compatible with all Ethereum network clients, and its functionality may be inadequate for developing complex smart contracts. Nevertheless, Brownie represents a useful tool for smart contract testing and development, enabling streamlined and efficient testing and deployment of smart contracts.

#### Foundry – Fuzzing

Foundry provides a powerful [fuzz](https://book.getfoundry.sh/forge/fuzz-testing) testing feature that can help developers identify vulnerabilities in their smart contracts. Fuzz testing involves generating random inputs and sending them to the smart contract to see how it responds. Foundry's fuzz testing module can automatically generate a large number of inputs and run them through the contract, recording the responses and identifying any potential issues. The tool can also generate inputs that are specifically designed to trigger edge cases or uncommon scenarios, allowing developers to test their contracts under a wide range of conditions. By using fuzz testing, developers can identify potential vulnerabilities that may not be detected by other testing methods, such as unit testing or manual code review. This can help to improve the overall security and reliability of smart contracts, ensuring that they are able to perform as intended in a variety of real-world scenarios.

### AI smart contract analysis

Current attempts at the use of ML techniques are mainly focused on the detection and classification of known exploits within a single function. This is inherently limited as none of these systems focus on the wider environment that it will be deployed into. They are only detecting if the function will run as intended not if the design of the overall system is flawed. The two papers focus on the same problem, the classification of vulnerabilities, they use [KNNs, SGD](https://downloads.hindawi.com/journals/scn/2021/5798033.pdf) and [Graph based methods](https://www.ijcai.org/proceedings/2020/0454.pdf), none of which are able to effectively evaluate the contracts vulnerability to external factors. The focus on the defence of the contract limits the capability of the use of ML methods. They have no goal, nothing to reach, without that they cannot achieve anything.

## Requirements and analysis

### Overview

The main goal of this project is to lay the foundational work for multi smart-contract exploit detection, this will be done by using graph-based methods to more efficiently search the possible space of combinations of contracts to make it computationally feasible to explore the space with reasonable certainty of it being free of any bugs or errors. The system should be able to explore a series of contracts, understand how they may be linked and by what means and generate a graph describing which to then be explored.

This project will consist of three main parts. The first task will be the creation of example contracts to demonstrate and build the system on. Building the implementation of the contract to graph converter will be the following task. The final task will be the running of the testing of said contracts with the now more optimised search space, this will then be used to review the improvement over random searching.

### Initial Analysis

#### Contract Creation

Example contracts will be required to test the system. Both in terms of its ability to generate the graphs representing them and ensuring that the intentionally engineered exploits are correctly programmed without any bugs. The contracts should cover a range of functionalities and complexities and should be designed to intentionally contain exploits or vulnerabilities to test the effectiveness of the system in detecting them. The contracts should also be well-documented, with clear explanations of their intended functions and any known vulnerabilities.

#### Contract to Graph Converter

The second task will involve building the implementation of the contract to graph converter. The converter should take the source code of the contracts and produce a graph representation that captures the relationships between the contracts and their functions. The graph should be designed to enable efficient search algorithms to be applied to it and should also be capable of being easily updated as new contracts are added to the system.

#### Testing and Review

The final task will be to run the testing of the contracts using the optimized search space generated by the graph-based methods. The results of these tests will be used to evaluate the effectiveness of the system in detecting exploits and vulnerabilities. The review will focus on the improvement in efficiency and accuracy compared to random searching, and any limitations or issues that arise during testing will be addressed and resolved. The goal is to develop a system that can provide a high level of confidence in the security of smart contracts and enable developers to build safer and more reliable blockchain applications.

### Summary of Requirements

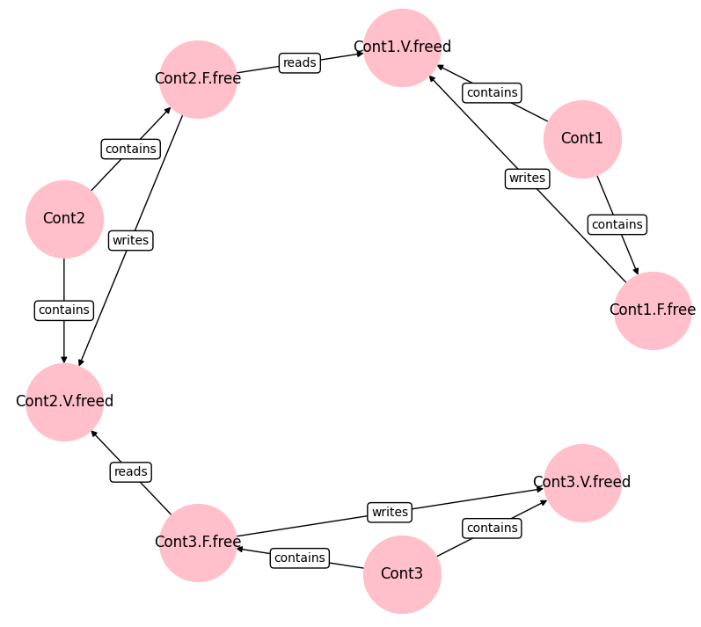
After analysing the literature review and considering potential limitations such as available technologies and time constraints, the essential features have will be implemented in the system along with their priority levels. Only after completing these essential requirements will, the desirable ones be completed.

|  |  |
| --- | --- |
| Requirements | Priority |
| Create a series of contracts that have a known exploit that can be used to demonstrate the system | Essential |
| Conversion of contracts into a graph-based structure with variables and function relations extracted | Essential |
| Generation of paths to inspect | Essential |
| Testing of paths to determine which are exploits | Essential |
| Reporting of results such that actional able changes can be made to improve the security of the contracts | Essential |
| Diverse types of contracts that can be supported the examples given are limited in their scope | Desired |
| Uses the ABI to generate the graph instead of reading the solidity code | Desired |

### Methodology

The solution proposed to address the target problem involves utilizing a parse tree that will be transformed into a graph, illustrating the interconnected nature of a collection of contracts. Subsequently, the contracts will be simulated to modify a target variable. This approach is promising in resolving the target problem as it effectively narrows down the search space by identifying the relevant functions and contracts linked together. Consequently, this strategy has the potential to enhance efficiency by streamlining the process of auditing a set of contracts.

A parse tree is used to “break” down the contract into its constituent parts, this involves data such as imports, variable names, and functions etc. This allows for a deep and rich representation of how the contract works, the tree can then be parsed into a graph of all the external parts of a contract. This includes things like variables and functions, meaning that we then can understand what can be called and what can be changed by anything external. Once the graph is generated it can be traversed to find paths that might be possible to exploit as instead of searching all the combinations, there are just a few paths to search down.



For example, the graph shown here that has been generated shows how three contracts can be connected, via the reading and writing of variables. As humans we can see that if we wanted to change Cont3.V.freed then that can trace back the different functions that might be associated with changing its value. Rather than searching blindly.

To compute the number of possible function calls you could use this function. Where *n* is the number of functions that affect the state of the EVM not just read a value. *r* is the number of functions that are being called, this allows it to span from 1 function call to all n.

Just for this small example there are already 15 different combinations, as you can see in the graphs below, the search space expands rapidly. For an [ERC-20](https://eips.ethereum.org/EIPS/eip-20#methods) contract where it only has 4 basic functions resulting in 64 different combinations. This does not even consider the potential for loops that result in multiple calls or even the possible number of inputs which is on average 2^256 per parameter.

Chart, line chart

Description automatically generatedChart, line chart, histogram

Description automatically generated

Chart, line chart

Description automatically generated After 100 functions there are more combinations than the number of atoms in the universe or possible magic cards in a commander deck.

#### IDK where this is supposed to go

In software development, it is important to ensure that resources are effectively managed and cleaned up after use to avoid potential issues such as memory leaks or other types of resource leaks. A context manager is a programming construct that provides a way to manage resources such as file handles, network connections, or database connections in a controlled manner. In the case of the creation and management of a graph, a context manager was utilized to ensure that the necessary actions were taken once the generation of the graph was completed. This approach helps to prevent issues related to resource management and ensures that the code is executed efficiently.

One of the benefits of using Python for contract analysis is that it can be seamlessly integrated within a Brownie test. Brownie is a Python-based framework for smart contract development that provides a range of tools and utilities to streamline the development process. By integrating contract analysis directly within a Brownie test, developers can more easily test their code and ensure that it is functioning as expected. This helps to streamline the development process, reducing the amount of time and effort required to write and execute tests on larger codebases. Overall, the use of Python and Brownie in contract analysis provides a powerful and efficient toolset for developers, enabling them to focus on writing high-quality code that meets their project requirements.

Non exhaustive list of references / list of useful links

1. Deep Thinking: Where Machine Intelligence Ends and Human Creativity Begins., Deep Thinking: Where Machine Intelligence Ends and Human Creativity Begins.
2. <https://arxiv.org/pdf/2302.07347.pdf>
3. <https://core.ac.uk/download/pdf/82416379.pdf>
4. <https://ethereum.github.io/yellowpaper/paper.pdf>