

# Holistic smart contract auditing

Thomas P Biddlecombe

Submitted May 2023, in partial fulfilment of  
the conditions of the award of the degree MSci (hons) Artificial Intelligence  
School of Mathematics, Computer Science & Engineering

Liverpool Hope University  
Supervisor: Dr Ogbonnaya Anicho

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature 

Date 12/05/2023

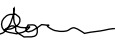
This dissertation is submitted in partial fulfilment of the requirements for the degree of MSci (hons) Artificial Intelligence offered by School of Mathematics, Computing and Engineering, Liverpool Hope University.

I confirm that this dissertation is my own work and wherever required I have acknowledged the work of others.

I confirm that I have obtained informed consent from all people who participated in this work and received ethics approval as appropriate to my dissertation.

I confirm that the word count for this dissertation including title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is 6,243 .

I also permit Liverpool Hope University to store a copy this dissertation in a public archive.

Signature: 

Date 12/05/2023

## Abstract

short and focused one page review of your work

## Acknowledgements

thanking anyone who helped you

Dr Ogbonnaya Anicho

# TODO

## Table of contents

### section (and sub-section) names and their page numbers

Abstract.....	2
Acknowledgements.....	3
Table of contents.....	4
Introduction .....	5
Literature review .....	8
Unit testing.....	8
Fuzzing .....	8
Static analysis .....	8
Formal verification .....	9
Machine learning approaches.....	9
Conclusion.....	10
Methodology.....	11
Requirements.....	11
System design .....	12
Tools and software used .....	13
Description of work and results .....	14
Overview of the solution.....	14
Contract Creation .....	14
Contract to Graph Converter .....	15
Exploit finding .....	16
Summary .....	16
Results.....	17
Comparison.....	18
Limitations.....	19
Strengths .....	19
Weaknesses.....	19
Conclusion.....	20
References.....	21
Appendices.....	22
Graphs about exploits .....	22
Contracts .....	23

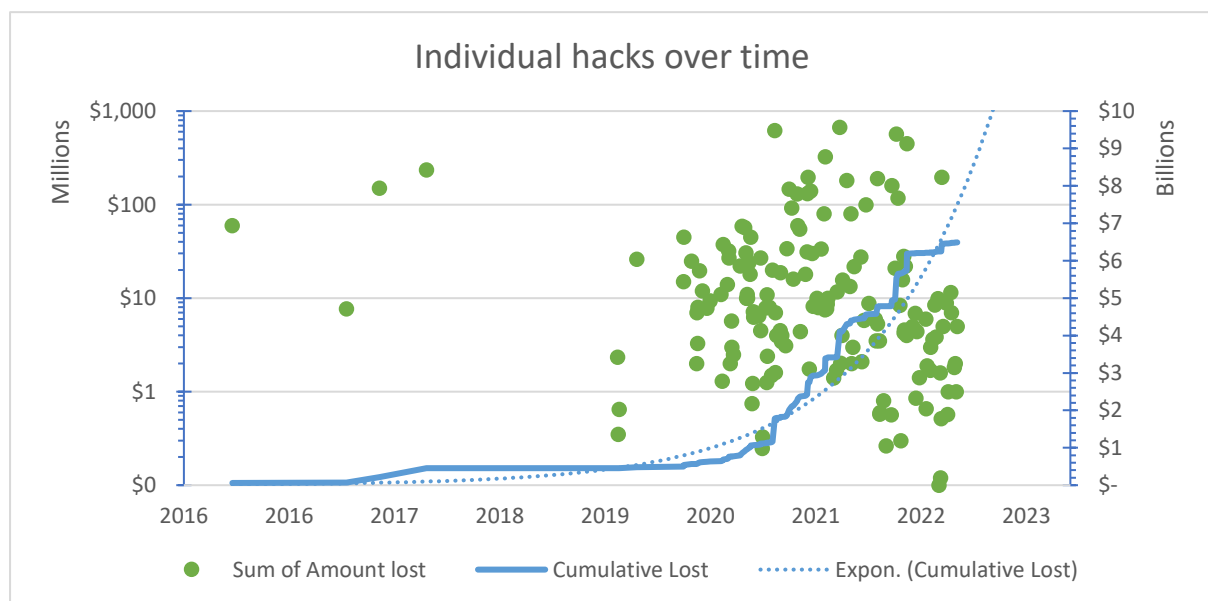
## Introduction

The Ethereum Virtual Machine (EVM) ecosystem faces a big problem which is that there are 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, because of the interconnected nature of the EVM, there are too many contracts in the networks that exist for an auditor to account for the myriad of ways that it might be interacted with. The ecosystem needs a better tooling which ensures a higher level of quality and security of contract.

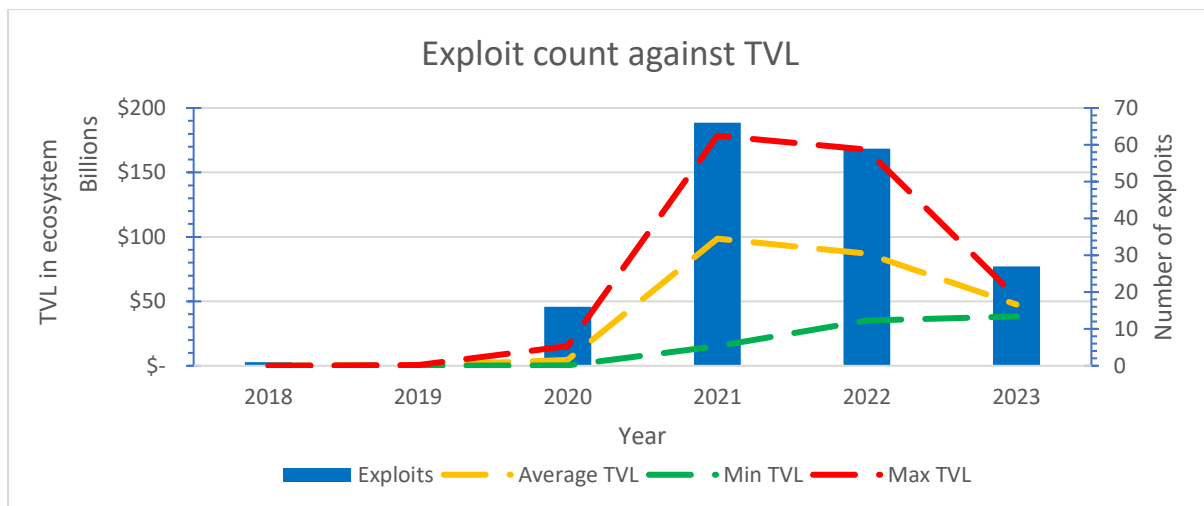
The reason why better tooling is needed is due to billions of dollars being lost every year to exploits. Although auditing of individual smart contracts has improved massively, there are no formal tooling for detection of exploits that would scan multiple contracts. Current tooling is limited in scope being unable able to take a holistic view and consider the wider interactions that a contract might have.

As the industry has evolved and matured there has been a sort of arms race between the auditors and their tools and those who which to exploit the contracts. Very rarely do you hear about a single contract being exploited, often the exploits being performed involve many contracts and it's their interconnected nature that makes them vulnerable. Where something that wasn't picked up in a single contract or there is a flaw in the economics, can become the seed for a wider exploit.

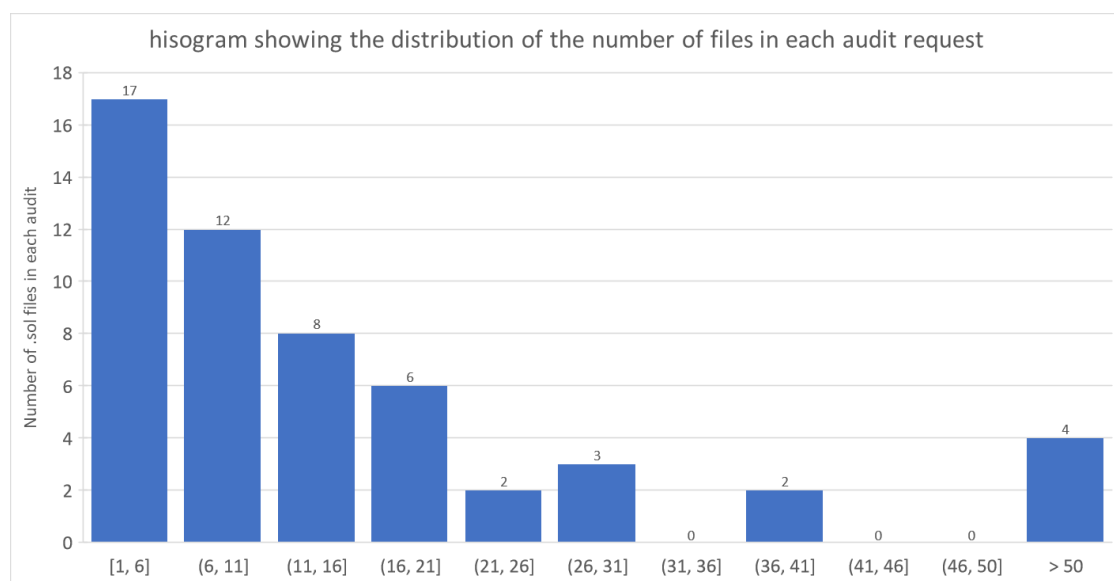
\$6.5 billion has been lost to exploits in the Ethereum ecosystem, at this rate the cumulative loss will reach over \$20B by 2024. This is unacceptable and would make many potential investors apprehensive to invest their money in the ecosystem. How can on average since 2020, 2.8% of all the ecosystem be lost to hacks each year.



The correlation coefficient between the average TVL and the number of exploits that occur is 0.98. The correlation between average TVL and the amount lost each year is 0.87 these two measurements and the graph shown below clearly show that the ecosystem is not getting any safer. This is deeply troubling as it shows that the ability of exploiters is matched against the ability of the tooling and auditors. In an ideal situation the number of exploits proportionally against the TVL would decrease, but unfortunately this cannot be observed.



The interconnected nature of smart contracts can be simply demonstrated here, this chart shows a histogram of the number of contracts being asked for audit. An argument could be made that an auditor could hold in their head and understand up to 10 contracts, but with one project having 90 contracts all being asked for an audit. There is no way that anyone can understand what is going on, this perfectly demonstrates why better tools that can understand and exploit the connections between contracts is so important.



The aim of this paper is to introduce a new technique for auditing a group of smart contracts and analysing their interactions collectively, as opposed to auditing each contract individually. Assuming that each contract functions correctly in isolation, it is possible to create an abstract representation that can be efficiently traversed, enabling the identification of potential vulnerabilities. Unlike most existing approaches that focus on detecting flaws in the code, this method adopts a goal-based approach, wherein a specific variable is provided, and the system attempts to find a series of contract calls that achieve the desired outcome. By considering the holistic relationships between the contracts, such a tool could become an asset in overcoming the impasse of the ongoing "arms race."

Another very important change that this approach allows is based in how it goes about finding the exploit, whereas most tools look for patterns and classify certain parts of code as vulnerable. The proposed approach will allow for a far richer representation of what qualifies as an exploit. There is no way for a static analyser for example to understand the richness and complexity of the contract it is looking at; this is because it must be programmed to find them as its rule based. Therefore, it

cannot find an exploit that it wasn't told to spot. In comparison the developer who's written these contracts, knows them better than anyone else, and at a far higher level of abstraction than any tool understands, "intended behaviour". This enables the developer to define a target and a way to change it that would be considered an exploit, and this enables the kind of richness in representation that this project aims to demonstrate.

In summary, this paper proposes a novel technique for auditing smart contracts by considering their collective behaviour. Instead of individually auditing each contract, the method employs an efficient search of the possible space to ensure comprehensive coverage. By creating an optimized abstract representation, the computational effort required to search for potential exploits is significantly reduced over brute forcing. Unlike traditional vulnerability-focused approaches, this technique takes a goal-oriented perspective, aiming to identify a series of contract calls that manipulate a given variable. With its ability to analyse the interconnections among contracts, this holistic approach has the potential to revolutionize the field and break through the current stalemate of the arms race in smart contract security.



## Literature review

analysing the current state of smart contract auditing

*this should not just be a list of reviewed papers, but a comprehensive and informative presentation of how reviewed works relate to your problem.*

This section will focus on the current state of smart contract auditing, its limitations and what it has achieved so far. With increasing complexity, going from Unit tests, through static analysis and formal verification to machine learning methods.

## Unit testing

Through unit testing, developers can test a broad range of contract features by simulating their behaviour in various environments, ensuring the identification and resolution of potential vulnerabilities before deploying the contract to the blockchain.

The problem with unit testing is that it is a highly manual and labour-intensive process that is primarily useful during the development of the contract. This is because when writing smart contracts, test driven development is a very commonly followed practice. Due to the manual portion of writing tests, the scope of them is limited, and although there are tools which can measure the coverage of the tests, not every possible combination of calls can be manually accounted for.

## Fuzzing

Fuzzing is a testing technique that extends unit testing by enabling the emulation of multiple cases instead of one. In contrast to fixed inputs with one expected output, fuzzing involves parameterizing inputs and expected results to generate hundreds of test cases. While fuzz testing can help improve the overall security and reliability of smart contracts, it has a limitation in that it cannot test every combination of contract calls.

Despite the limitation of fuzzing, it remains a valuable testing technique for identifying potential vulnerabilities that may not be detected by other testing methods. By automatically generating inputs and running them through the contract, developers can identify vulnerabilities and make the necessary changes to improve the security and reliability of their contracts. However, it is important to note that fuzzing and unit testing are not exhaustive, and there may be some scenarios that are not thought of to be tested.

## Static analysis

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 reentrancy 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 several 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. Being rule based and rigid it cannot find exploits that are just outside of its rule set, limiting its ability to generalise leading to potentially missed exploits. 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.

### Formal verification

Formal verification is a critical process for ensuring the reliability and security of smart contracts. It employs mathematical and logical techniques to rigorously analyse and validate contract behaviour, aiming to detect and prevent potential flaws or vulnerabilities. Researchers have explored various methods and tools to enable formal verification, systematically examining contract properties such as safety, functional correctness, and adherence to specified requirements. These approaches provide a solid foundation for enhancing the trustworthiness and dependability of smart contracts in the context of blockchain technology.

The verification of smart contracts has gained significant attention in recent years due to the growing importance of blockchain technology and the potential risks associated with the execution of flawed contracts. Several researchers have explored different approaches to formally verify smart contracts and address the security challenges they pose. One notable line of research focuses on formal verification using programming languages and tools. For example, Bhargavan et al. (2016) proposed a framework to analyse and verify the safety and functional correctness of Ethereum contracts by translating them to F\*, a functional programming language aimed at program verification. Their approach utilizes automated queries for an SMT solver to statically verify properties of contracts, considering both the high-level Solidity source code and the low-level EVM bytecode.

However, one of the key challenges in smart contract verification lies in the interactions and compositions of multiple contracts. Smart contracts in Ethereum can call into the public methods of other contracts, which introduces complex dependencies and potential security risks. Verifying the correctness and safety of multiple contracts together becomes a challenging task due to the decentralized and interconnected nature of the Ethereum network. The interconnectedness creates a need to reason about the behaviours and interactions of multiple contracts, ensuring that they collectively behave as intended and do not introduce vulnerabilities. Addressing the verification of multiple contracts requires advancements in verification techniques, including the ability to reason about contract interactions, data flows, and potential security issues that may arise from their composition. Future research in this area should aim to develop more comprehensive approaches that can analyse and verify the correctness and security of multiple contracts in a decentralized environment effectively.

### Machine learning approaches

Current attempts at the use of ML techniques are 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. The main advantage over rules-based methods like static analysis is that they might be able to detect more obscure versions of known vulnerabilities that are slightly outside of the training set.

### KNNs

# TODO INTRO

used [Oyente](#) and [SmartCheck](#) instead of something more popular and being actively developed like [Slither](#). Smart check has been deprecated since 2020 and Oyentes last commit was back in 2020 prior to the papers publish. The use of inferior tooling and the omission of tools like Slither might indicate

that their methods might not be as effective. For example, the tool [SolidiFi](#) used an AST to inject vulnerabilities into code the ran the dataset on each tool evaluating their results. Slither demonstrated exceptional performance, accurately detecting reentrancy, timestamp, and tx.origin bugs with nearly 100% accuracy. Where as the KNN only has a 95%, N/A and 95% accuracy in detecting the respective vulnerabilities. SolifiFi was released before the KNN paper so they could have used that to test their dataset more easily on a wider range of static analysers.

### [Graph Based Methods](#)

#### # TODO INTRO

worked by converting the code into a directed graph with the edges and nodes being typed, this graph was then converted into a matrix where a KNN would then perform classification on it. The scope of this project is more limited as it is only focusing on reentrancy, timestamp and infinite loops, its output is only limited to a binary classification of vulnerability or not which is further limited in scope again (See Fig .2 from the paper). So its not even predicting which vulnerability occurs just if there is on which would not help a developer. Although the TMP model the paper discusses does achieve an average accuracy of 84% in reentrancy and timestamp dependence vulnerabilities. This is still not as good as tools like Slither as discussed earlier.

### Summary

Current attempts at the use of ML techniques are 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](#) and [Graph based methods](#), none of which can 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.

The papers discussed both fail to clearly demonstrate an improvement over static analysers they also fail in addressing the problem which this paper aims to solve. They are both still limited to the base contract level, focusing on individual functions unaware of how the contract is intended to be used and

### Conclusion

These are not enough, clearly, as discussed in the introduction, things are not improving. Even though these tools do ensure a certain level of security, much like a seatbelt it only keeps you safe if you use it. I've personally observed projects being deployed without audit where it turns out that they haven't even written a test script, and the static analyser that's built into the IDE that they were using was showing errors leading to a \$1.5M bug that I had to hot fix. If the project had gone to audit the tools above would have most likely been used by auditors, but these should have been run by the local development team which would have picked up the errors avoiding the situation entirely.

## Methodology

the approach you use to solve the problem in terms of tools and technologies.

This section will discuss three main areas of the project, the requirements, which details what needs to be developed to ensure that the project is completed. The system design which will layout a plan and discuss the pros and cons of certain design decisions. The tools and software used, following from the system design, this section will provide an overview of the tools and how they will aid in the development of the project.

### Requirements

After analysing the literature review and considering potential limitations such as available technologies and time constraints, the essential features have will be implemented. 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
Simple reporting of results	Essential
Reporting of results such that actional able changes can be made to improve the security of the contracts	Desired
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

## System design

This section is split up into three parts, Contract creation covering how the inputs to the system will be created. Contract to graph converter, this covers how the contracts from the previous step will be abstracted and converted into a graph. Exploit finding, this covers how the graph will be traversed as the solution attempts to find the exploit. These three sections will make up the primary part of the project.

### 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.

Having a thoroughly tested set of contracts which can be relied upon is important. This is because the environment in which the project will be run in must be as closely representative of how it would be deployed in production. This is important because if the environment cannot be relied upon then the results from the test cannot either, this would call into question any findings that this paper presents.

### Contract to Graph Converter

The second part will involve the building and 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 will be designed to enable application of efficient search algorithms and will also be capable of being easily updated as new contracts are added to the system. These properties are important as ease of development will ensure code readability and maintainability of the code base. Having the graph structure be easily traversable while maintaining the rich structure and information it holds will ensure faster compute times for the different paths.

### Exploit finding

The final part 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.

The final part is the running of the graph to generate the paths which are then tested. The paths are generated from an adapted version of Breath First Search (BFS) where instead of just yielding the nodes it returns the full path and handles loops. BFS was chosen for the main characteristic of finding the shortest path length. To test the multiple paths that are returned, the test network that the contracts have been deployed on will be reset each time to just after the contracts have been deployed ensuring that the state is the same each time meaning that each run wont have nay external impact on the validity of the results.

## Tools and software used

In this section, we will discuss the tools and software used in the project. Hardhat is a popular Ethereum development environment that simplifies the compilation, editing, debugging, and deployment of smart contracts. Solidity Parser converts Solidity files into abstract syntax trees (AST), allowing for an object-oriented representation of contracts and the creation of a graph to visualize their relationships. NetworkX is a versatile network analysis tool used for graph creation, analysis, and visualization. Eth-Brownie, with its integration with Python, is employed for exploit detection and interaction with a locally hosted network. Hardhat is the preferred choice for contract development, while Brownie is used specifically for exploit detection.

### HardHat

[Hardhat](#) is a widely used Ethereum development environment for the EVM, that aids in the compiling, editing, debugging and deployment of smart contracts. It's ease of integration with JavaScript and Type-Script testing frameworks makes it a very popular choice for developers. Due to Solidity being a tightly defined language, using TypeScript over JavaScript or Python is a clear and sensible choice that many developers make when writing unit tests.

### Solidity Parser

The Solidity-Parser developed by [ConsenSys](#) then bug fixed further by [florian1345](#), allows for the conversion from a solidity file to an abstract syntax tree (AST) which can then be referenced in an object oriented fashion. This is useful because it allows for a more easily abstracted view of the contracts which can then be joined together in a graph, linking the relationships between them all. This method is inherently limited as it only supports contracts written in Solidity. The way to circumvent this is by analysing the ABI instead, but as this is only supposed to demonstrate the proposed method, spending the extra time to implement this is not conducive to further proving the method.

### Network X

[NetworkX](#) is a wide-ranging network analysis tool, this allows the creation, analysis, and visualisation of graphs. This can be utilised to represent the graph showing the connections between many contracts. By wrapping a custom class to handle graph management in a context manager, NetworkX can be used in a highly abstracted way, which enables a highly readable codebase.

### Eth-Brownie

[Brownie](#) is another smart contract tool but with only a subset of the functionality of HardHat. Due to it being integrated with python, a weakly typed language, it is unsuitable to develop and test smart contracts in. But, because of it easily integrating with python, its use in the project in a limited scope is acceptable. The main feature being utilised is its ability to run a locally hosted network and interact with it all from python. Another benefit is being able to deploy compiled smart contracts all within the python script also. For these reasons it is preferable to use Brownie instead of HardHat for the exploit detection, but not for the development of the test contracts.

## Description of work and results

describe your work in details and the results you get.

### Overview of the solution

The solution proposed aims to address the target problem, this involves utilizing a parse tree which will be transformed into a graph, this illustrates the interconnected nature of a collection of contracts. Subsequently, the contracts will be simulated with the goal of modifying 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 being linked together. Consequently, this strategy has the potential to enhance efficiency by streamlining the process of auditing a set of contracts.

### Contract Creation

#### Environment setup

Using HardHat, the contracts can be written and tested to ensure that the intended exploit does exist, and any other unintended methods are not possible. This is important for two reasons; unintended vulnerabilities mean that the testing of the contracts will not be reliable. Secondly The contracts need to be compiled to enable Brownie to deploy the contracts and their bytecode to the test network so that they may be tested. Both HardHat and Brownie are industry standard tools that have been thoroughly tested.

#### Contracts overview

There are three contracts that make up the exploit, the rough structure is as follows; It stores the previous contract and a boolean. If the previous contracts boolean is true, then this one can become true when *free* is called. This will continue up the chain until the target variable has been flipped.

```
import {foo*} from "../PreviousContract.sol";

contract foo{
    freed = false;
    cont;

    constructor (address _foo*Address) {
        cont = foo*(_foo*Address);
    }

    function free () public {
        require(cont.freed() == true, "foo* is not freed");
        freed = true;
    }
}
```

The reason this structure is chosen is because all auditing tools that are available find would consider this to be safe. This is only when viewed singularly, but when deployed the weakness is glaringly obvious. For example, the Remix IDE's static analysis cannot find the vulnerability and the only security concern is a reentrancy which upon further inspection is a non-issue. Another reason is that the current tooling doesn't allow for users to input what should and should not be allowed to be changed.

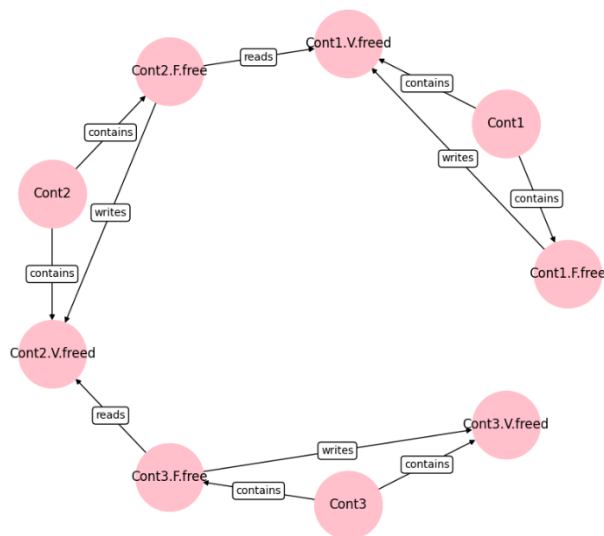
## Contract to Graph Converter

### Abstract Syntax Tree

An AST is used to “break” down the contract into its constituent parts, this involves data such as imports, variable names and functions. Allowing 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 variables and functions, meaning that a representation can be formed of what can be called and 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.

### Graph Generation

For example, the graph shown here that has been generated shows how three contracts can be connected. 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.

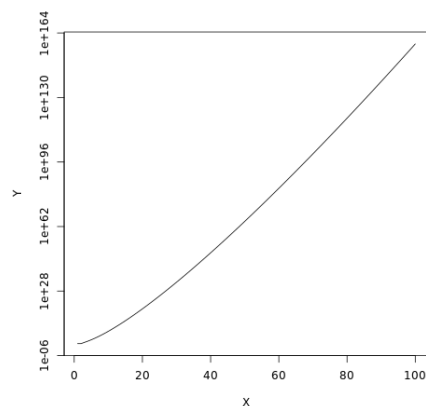


### Search Complexity

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. Where  $r$  is the number of functions that are being called, this allows it to span from one function call to all  $n$ .

$$f(n) = \sum_{r=1}^n \frac{n!}{(n-r)!} \rightarrow \text{number of combinations}$$

Just for this small example there are already fifteen different combinations, as you can see in the graphs below, the search space expands rapidly. For an [ERC-20](#) 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. After one hundred functions there are more combinations than the theorised number of legal chess moves.





## Exploit finding

### Contract deployment

The next step is to match the compiled contracts with the analysed ones from the repository. This enables them to be used inside of a test script where the graph is being traversed and thusly needs to know what each node corresponds to. Returning a dictionary of contract names (as known by the graph) and Brownie contract objects this means that it is quite easy to reference the intended object. Meaning that using the snapshot and revert features, resetting the network, which can be set to just after the contracts were deployed, resulting in it being faster and simpler to try lots of different paths.

### Graph Traversal

One of the benefits of using Python for contract analysis is that it can be seamlessly integrated within a Brownie test. With the dictionary holding the pairs of the contracts and their names, the graph can then be traversed. Brownie allows the use of checkpointing and reversion to said checkpoint meaning that once the contracts have been deployed each test run can then be reverted to the checkpoint afterwards. With the checkpoint being placed just after the contracts have been deployed, revision to that state is easy and ensures a reliable testing environment.

### Path Generation

The generator takes in the graph and uses recursion to traverse said graph, yielding every path leading from a source node outward. This approach of yielding and generating the paths will ensure memory availability instead of searching out to a depth of  $n$  nodes which could be in the order of billions of paths, there is no need to compute them all at once. As an adaptation of breadth first search, where instead of returning the nodes, it needs to also return the path it took.

```
graphTraversal(currentNode, maxDepth):  
edges ← []
```

```
FOR EACH edge IN edges WHERE currentNode IN edge THEN:
```

```
    IF currentNode IS variable THEN
```

```
        edges.append(nextNode)
```

```
    ELSE IF currentNode IS function THEN
```

```
        edges.append(nextNode)
```

```
FOR EACH edge IN edges:
```

```
    IF edge IS NOT variable THEN
```

```
        yield temp
```

```
IF maxDepth IS 0 THEN
```

```
    RETURN
```

```
FOR EACH edge IN Edges:
```

```
    a ← graphTraversal(currentNode + [edge], maxDepth - 1)
```

```
    FOR EACH _x IN a:
```

```
        yield _x
```

## Summary

The proposed layout and structure of the project will enable the compilation of the contracts to get their ABI. The abstraction of their solidity code and linking into a graph will allow for a better and

more richly represented structure of the project. The close integration of Brownie will make testing of the different paths simpler and highly repeatable and reliable. These three main factors enable the project to achieve its goal of demonstrating a novel approach to smart contract analysis.

## Results

Below is a rough and simplified output of what running the program does, once the contracts have been deployed, the generator of the different paths will start yielding and those paths can be tested. Starting from the target contract and moving down the list, it only takes 3 tries to get the correct path, which is then returned to the main body and printed out to the user.

### Contracts are deployed

```
Deployed: cont1 from ../contracts/artifacts/contracts/Cont1.sol/Cont1.json
at 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87
Deployed: cont2 from ../contracts/artifacts/contracts/Cont2.sol/Cont2.json
at 0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6
Deployed: cont3 from ../contracts/artifacts/contracts/Cont3.sol/Cont3.json
at 0xE7eD6747FaC5360f88a2EFC03E00d25789F69291
```

### The first path is tested

```
Testing new path
Testing Path: ['cont3.f.free']
Calling cont3.f.free
Transaction reverted (Cont2 is not freed) The path is unsuccessful
```

### The second path is tested

```
Testing new path
Testing Path: ['cont2.f.free', 'cont3.f.free']
Calling cont2.f.free
Transaction reverted (Cont1 is not freed) The path is unsuccessful
```

### The third path is tested

```
Testing new path
Testing Path: ['cont1.f.free', 'cont2.f.free', 'cont3.f.free']
Calling cont1.f.free
Transaction confirmed

Calling cont2.f.free
Transaction confirmed

Calling cont3.f.free
Transaction confirmed The path is successful
```

### The order of function calls is displayed

```
Found path: ['cont1.f.free', 'cont2.f.free', 'cont3.f.free']
```

The result of the graph traversal method was that the exploit was found on the third attempt whereas it could have taken up to fifteen tries, with an expected count of 8 tries. If the chain had a length of four its max complexity would be sixty-four instead of the four tries that it would take to find the exploit. For more complex graphs my method's complexity would be exponential at worst, and linear at best, in comparison to factorial. This is a huge efficiency gain over brute forcing without any direction demonstrating the possible effectiveness of this approach to helping audit smart contracts.

## Comparison

### Static analysis

[Remix IDE's static analysis](#), was unable to find the exploit, this is because it is inherently limited in scope, only looking at a single contract. This is the exact problem that my solution is trying to solve, by better understanding the overall set of contracts and being given a goal. It will be able to find exploits that are currently not automatically detectable.

### Manual Unit testing

The only other way to detect a multi-contract exploit is by using manually written unit tests, this is inherently manual, slower, and limited by the skills of the person writing them. There is no guarantee that an auditor will be able to find them, when there are multiple contracts involved, being able to understand how they all interact.

### Summary

The goal of this project was to demonstrate a new and novel approach to smart contract auditing that would be able to detect new vulnerabilities by focusing on the interconnected nature of smart contracts. The solution that has been outlined, developed, and evaluated clearly shows how it unlocks a whole new dimension of smart contract auditing. This allows two main things, enabling auditors to better understand the interconnectedness of a project and be able to receive it with a certain level of confidence that it is secure. With the abstraction of the contracts in a project, a holistic view can be taken this allows auditors to more quickly see how each contract interacts with one another ensuring that connections don't get missed. Secondly, the quality of the contracts can be held to a higher level, using static analysis, problems within individual contracts can be checked. But the overall interconnected nature of the contracts needs to be checked over and the solution I'm proposing will help to ensure a higher quality of contracts.

## Limitations

describe the strengths and weaknesses of your work

## Strengths

# TODO

## Weaknesses

Due to the aim of the project being to demonstrate a solution to the problem discussed, it is more limited in scope in two main ways. Firstly, because an AST is used, its very easy to pull out the features from a set group of contracts but would not be suitable for deployment. Instead, analysing the ABI directly, although it would be working on a more abstracted problem. It would make the extraction of the links far more generalisable and resilient to any contract. Secondly, the problem of parameters is not tackled, for example on a simple token transfer there are  $2^{256}$  different numbers to pick from, this is handleable by following the similar method used by Foundry in its fuzzing module, by inputting powers of two up to the max value of 255 you can simulate a significant range of numbers.

## Conclusion

*this should include a summary of your findings, if required a comparison with any previous works and anything else you think is important to include in this section*

This project has the potential to have a significant impact on the auditing process. The use of a parse trees to generate a graph of the contracts allows for a more streamlined and efficient approach to identifying potential exploits that cannot be identified by traditional means. By narrowing down the search space and identifying the relevant functions and contracts linked together, this strategy could enhance the efficiency of auditing a set of contracts.

Furthermore, the project sheds light on the complexity of the search space for potential exploits, highlighting the need for a more efficient approach such as the one proposed. Overall, the project has the potential to enhance the security of smart contracts by providing a more efficient and effective means of identifying potential vulnerabilities. This is not going to replace traditional auditing but might be able to reduce the workload of the auditing teams as more of the contracts will be checked over prior to it going to audit

# TODO

## References

*Here you should list the papers, articles, books and other publications that you refer in the text in a standard IEEE Referencing System, please see this link for more information.*

Non exhaustive list of references / list of useful links

Deep Thinking: Where Machine Intelligence Ends and Human Creativity Begins., Deep Thinking: Where Machine Intelligence Ends and Human Creativity Begins.

<https://arxiv.org/pdf/2302.07347.pdf>

<https://core.ac.uk/download/pdf/82416379.pdf>

<https://ethereum.github.io/yellowpaper/paper.pdf>

<https://app.sherlock.xyz/audits/contests>

<https://defillama.com/>

<https://dl.acm.org/doi/pdf/10.1145/2993600.2993611>

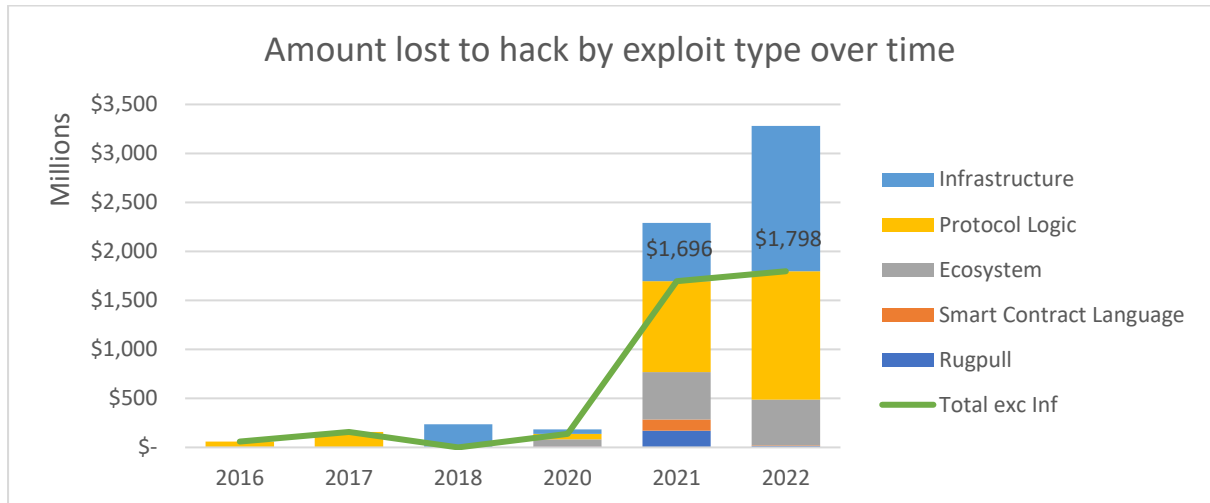
# TODO

## Appendices

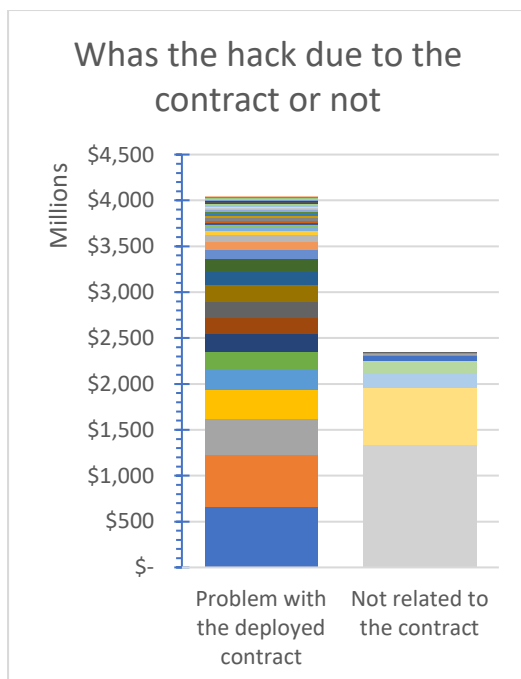
This is optional however you can include code, data, test data, manuals etc

### Graphs about exploits

Amount lost to hack by exploit type over time



Was the hack due to the contract or not



## Contracts

### The smart contract Cont1.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

contract Cont1 {
    bool public freed = false;
    constructor(){
    }

    function free() public{
        freed = true;
    }
}
```



### The smart contract Cont2.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import {Cont1} from "./Cont1.sol";

contract Cont2 {
    bool public freed = false;
    Cont1 public cont1;

    constructor( address _cont1) {
        cont1 = Cont1(_cont1);
    }

    function free() public{
        require(cont1.freed() == true, "Cont1 is not freed");
        freed = true;
    }
}
```

### The smart contract Cont3.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import {Cont2} from "./Cont2.sol";

contract Cont3 {
    bool public freed = false;
    Cont2 public cont2;

    constructor( address _cont2){
        cont2 = Cont2(_cont2);
    }

    function free() public{
        require(cont2.freed() == true, "Cont2 is not freed");
        freed = true;
    }
}
```