DEPARTMENT OF COMPUTER SCIENCE

Generalised Front-running Attacks in Blockchain

Building, formalising and mitigating generalised front-running techniques in blockchain environments.

Zachary Stucke

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Wednesday 11th May, 2022

Summary

Abstract

Front-running occurs when an agent is attempting to complete a financial transaction, an attacker then identifies this operation and executes their own replacement before the agent's can complete, usually profiting in the process. In blockchain environments, transaction information is publically available to all users before they are filled in the mempool; many cases and forms of front-running arise in this environment. The public nature of these pending transactions allows capable attackers to implement these attacks.

In this project, I outline, implement and mitigate some common forms and methodologies of front-running in Ethereum-based networks. I build a system for interacting with Ethereum blockchains using asynchronous Python and Web3.py. I design two smart contracts with the Solidity programming language with front-running vulnerabilities. I implement the front-running types: Displacement, Sandwich and Priority Gas Auctions (PGAS) using both the real Goerli test-net and the local Ganache blockchain. I provide a dashboard for users to run these attacks themselves at http://3.66.150.226/. I also implement and visualise the new MEV-geth method for sending transactions on the dashboard, showing it can provide mitigations against some of these attacks.

Achievements

- I designed and implemented a system for highly performant, asynchronous blockchain interactions with Ethereum-based networks in Python. Among more, users can access their wallets, send transactions, deploy and interact with smart contracts.
- I designed and implemented a local transaction pool and monitoring system to automatically manage multiple in-flight transactions and rebroadcast transactions when needed asynchronously during software execution.
- I designed and implemented a transaction simulation methodology that allows users to simulate transactions easily with the Ganache local blockchain. This functionality allows less established users to easily and efficiently simulate transactions in real-time without the need to run their own costly node. This also proves the barrier to entry for front-running is reasonably low.
- I researched, formalised and implemented generalised Displacement front-runners, Sandwich front-runners and Priority Gas Auctions (PGAs) to prove their ease of implementation in real blockchain environments.
- I researched and implemented a method to send Flashbots transaction bundles to MEV-geth miner implementers, allowing transactions to skip the pending transaction pool and prevent front-running attack vulnerability from non-miner searchers.
- I developed 2 Ethereum smart contracts using the Solidity programming language: 1. a simple holder contract where an attacker can execute a Displacement attack; 2. a simple liquidity pool contract where an attacker can execute a Sandwich attack. These contracts were used to create experimental environments to carry out front-running attacks safely.
- I integrated the system with an existing web framework, using Django as backend and React as front-end, to create a dashboard at http://3.66.150.226/ for users to experiment with real-world front-running examples. Users can run different front-running

experiments and visualise the transactions, balances and outcomes on both the agent and attacker. Users can run the experiments with the traditional or MEV-geth execution types, showing that the latter prevents the attacker from stealing funds. Displacement and Sandwich experiments were implemented on the real Goerli test-net, allowing the user to see their experimental transactions directly on the Goerli Ethereum Explorer; Priority Gas Auctions (PGA) are run on locally spun Ganache blockchain instances to save on gas fees.

- I wrote over 4500 lines of Python, over 1000 lines of Javascript and 138 lines of Solidity smart contract code to implement the blockchain interaction software, experiments and publically viewable dashboard at http://3.66.150.226/.
- I implemented 11 unit/experiment tests using the ganache blockchain to validate both the system and experiment functionality.
- I undertook a presentation to the Financial Technology with Data Science MSc cohort at the University of Bristol to introduce my work and resultant dashboard to validate its effectiveness as an informative and educational tool. This presentation is viewable on YouTube at https://youtu.be/AQ_8G7B1Q7Q.

Dedication and Acknowledgements

I would like to thank my Supervisor, Dr John Cartlidge, and my Shadow-Supervisor, Theodoros Constantinides, for their invaluable advice and shared experience that helped me produce this work.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Zachary Stucke, Wednesday 11th May, 2022

Contents

List of Figures

List of Tables

Ethics Statement

An ethics application for this project was reviewed and approved by the faculty research ethics committee as application 10761.

Supporting Technologies

A summary of relevant third-party resources used during the project:

- I used a web template framework built by myself, Zachary Stucke, **previously before this project**. This template implements management scripts, a React front-end with generic components, global store and request handlers, utilities and admin interface, a Django backend with generic database structures, API handlers and utilities. It also implements a task management system using Celery, allowing scripts to run outside the usual web request-response cycle.
- I used Python (https://www.python.org/) and Asyncio (https://docs.python.org/3/library/asyncio.html) for the asynchronous software implementation.
- I used the Web3.py Python library as the foundation for my interactions with the blockchain network obtained from https://github.com/ethereum/web3.py.
- I used the Flashbots Web3.py polyfill from https://github.com/lekhovitsky/web3-flashbots to add core functionality to Web3.py, allowing interactions with the MEV-geth *eth_sendbundle* and *eth_callbundle* RPC endpoints.
- An Infura node was used to connect my applications with Ethereum main and test networks. Infura is a cost effective way of connecting to the network without the cost or complexity of running a personal node. Homepage: https://infura.io/
- The Etherscan API, https://etherscan.io/apis, was utilised to identify historical transactions for a specific account; this functionality was not available through the standard Infura node used.
- I made heavy use of Ganache CLI, the local blockchain environment. During testing, Ganache was used to create specific scenarios and test interactions. In production, it was used to fork from real networks to run simulations during generalised front running attacks. Found at https://github.com/trufflesuite/ganache.
- The Goerli Ethereum test-net was used for the majority of experiments and transactions due to the implementation of the flashbots MEV-geth miner and the (almost) free transaction fees. Goerli test-net: https://goerli.net/.
- The Flashbots Goerli test-net validator/miner was used to test MEV transactions in a near cost-free environment. Unlike normal validators, this miner specifically runs the MEV-geth miner software; this software implements the RPC standard superset containing *eth_sendbundle* and *eth_callbundle* endpoints. Organisation: https://docs.flashbots.net/, endpoint: https://relay-goerli.flashbots.net.
- I used Amazon Web Services (AWS) to deploy my dashboard to an EC2 instance with an Elastic static IP. Found at https://aws.amazon.com/.
- I used Microsoft Forms to create and distribute a feedback form analysing the value of the created dashboard during a presentation. Found at https://forms.office.com/.
- I uploaded videos to YouTube that pertained to the project allowing easy access and viewing. Found at https://youtube.com.
- I used a GitHub repository to store and develop the project software. Found at https://github.com/.

Notation and Acronyms

ABI	:	Application Binary Interface
AMM	:	Auto Market Maker
API	:	Application Programming Interface
AWS	:	Amazon Web Services
BEV	:	Blockchain Extractable Value
CLI	:	Command Line Interface
DApp	:	Decentralised Application
DeFi	:	Decentralised Finance
DEX	:	Decentralised Crypto Exchange
EC2		Elastic Compute Cloud
ECDSA	:	Elliptic Curve Digital Signature Algorithm
EIP	:	Ethereum Improvement Proposal
ETH	:	Ether
EVM	:	Ethereum Virtual Machine
FIFO	:	First In First Out
HFT	:	High-Frequency Trading
HTTPS	:	Hypertext Transfer Protocol Secure
ICO	:	Initial Coin Offering
IO	:	Input-Output
JSON	:	JavaScript Object Notation
MEV	:	Maximal Extractable Value
NFT	:	Non-Fungible Token
PGA	:	Priority Gas Auction
PoS	:	Proof of Stake
PoW	:	Proof of Work
RDS	:	Relational Database Service
RPC	:	Remote Procedure Call
TX	:	Transaction
UML	:	Unified Modeling Language

Chapter 1

Introduction

1.1 Financial Markets

Financial markets are marketplaces where agents interact with one another to exchange assets; the value of an agent's assets change depending on both their own, and other agents' interactions with the market.

In most financial markets, agents act in their own best interest to maximise the value they can extract from their activities. Most agents do this on a level playing field, competing with one another for a limited set of resources. However, some agents have an advantage in the market over others: they may have access to extra information or access to tools and mechanisms that others do not. Examples in traditional financial markets include High-Frequency Trading (HFT), legal in most markets, and insider knowledge, illegal in most markets. The legality and ethics of different advantages vary dramatically depending on the country, market and type of advantage.

1.2 Blockchains

Blockchain-based systems have many similarities with traditional financial markets: agents exchange assets in the form of transactions and their assets' value depend on the actions of others. Blockchain-based systems are usually decentralised and information is usually publically available; many instances of illegal insider trading in centralised exchanges are intrinsically publically available in decentralised ones: the vulnerability is still there but is democratised to all agents. These systems both solve and suffer from some traditional exploits, whilst also opening the door to some new ones. The Ethereum blockchain, Ethereum Virtual Machine (EVM) and its corresponding easy to use programming language, Solidity, pioneered innovation and progress of blockchain-based systems [15]. However, this led to an increase in the complexity of the market and opened up a litany of new types of exploits available to attackers.

1.3 The Democratisation of Blockchain DApp Creation

Ethereum's smart contracts are applications built on the blockchain; they inherit the finality of operations usually present in blockchain environments. These contracts can be created by all actors in the system; Ethereum was the first blockchain to provide the ability for agents to quickly and cheaply create applications that were secured by a blockchain. Smart contracts led to the explosion in the number and complexity of decentralised applications (DApps) the world sees today.

Humans are notoriously bad at bug-proofing their software. Given the finality of blockchain environments and the ability for anyone to create smart contracts regardless of ability or scale, the number of bugs and exploits existing in DApps increased dramatically. This led to an increase in the number of atomically profitable opportunities available to actors. When one of these opportunities becomes known to an attacker, there is an incentive for the attacker to attack and extract the value from the exploit as fast as possible before any other agents are able to do so. The decentralised and publically available information in blockchain makes these attacks unusually easy to execute.

1.4 Motivation

Attacks and exploits in blockchain environments were initially theoretical however, in recent years, their execution has been observed and documented in the real-world [48]; the increasing prevalence of these attacks is causing genuine concern over the fundamental security of blockchain-based systems and agent's concern over the security of their transactions.

Agents who have experienced these attacks, or are worrying about them occurring, are looking for methods to protect themselves and confirmation that a mitigation they are implementing is sure to protect them. In this paper, I explore some of the common attacks that occur and mitigations available. I also implement these attacks and show that mitigation techniques available actually work in practice.

The majority of agents' activities as individuals do not directly affect that of other agents, outside of causing market moves. In this paper, I specifically focus on attacks by one agent that directly diminish the profitability/holdings of another agent. Currently, the legality of such activities seems to fall into a grey area in most jurisdictions [31] however, it's clear these attacks do occur and are a real issue for agents. Mitigations of such attacks will be explored.

Through implementing these attacks, I will have to create smart contracts that contain vulnerabilities to these attacks. Understanding these vulnerabilities is beneficial to other developers building their own DApps, helping them build their software in a more secure fashion.

1.5 Aims & Objectives

1. To build software to interact with any Ethereum-based blockchain: a system to send and manage blockchain transactions and accounts; deploy and interact with smart contracts; view and analyse public blockchain information and any other tools needed for a complete system.

2. To research different blockchain-based attacks and exploits that exist today, describe and formalise examples of them.

3. To implement some of these different blockchain-based attacks and simulate their workings in a safe environment.

4. To research and implement mitigations available to prevent an agent from falling victim to one of these exploits.

5. To provide a web dashboard to simulate some of these exploits, along with an execution method that implements mitigations preventing the attack from occurring. This dashboard should provide education and information relating to the implemented front-running attacks and how agents can protect themselves against these attacks.

Chapter 2

Background

2.1 Blockchains

2.1.1 Overview

A blockchain is a chained historical and immutable ledger of information and records shared between multiple distributed systems (nodes) worldwide. These nodes are inherently distrustful and self-serving however, through the cryptographic security of the network, they are able to work together to maintain a joint state and agreed history for actions taken on their blockchain. Blockchains have been theorized and developed incrementally over the last 50 years [53]. The largest, most successful and famous of which is Bitcoin, described and invented by the collective known as "Satoshi Nakamoto" [45].

A blockchain is made up of individual blocks, similar to a linked list. Each block is cryptographically connected to the previous block and the next in the chain, except for the head (most recent) and tail (root or genesis) blocks which are connected with only one other block. The blocks are connected with a hash of the data in the block combined with the previous block's hash; changing a piece of information in a block would change the block hash and all blocks ahead of it in the chain. As more blocks get added on top, it becomes increasingly computationally expensive (and eventually infeasible) for someone to change that information, which provides the system with its security and immutability so imperative to decentralised systems.

"Miners" are a subset of nodes that compete to "mine" the next block in the chain for a financial reward. The process by which they do this varies between blockchains: the most common of which are Proof of Work (PoW) [32] and Proof of Stake (PoS) [39]. Once a miner successfully mines and creates a new block, it broadcasts it throughout the peer-to-peer system, all other miners then start working on mining a block to append on top of the newly minted one.

When a disagreement occurs between nodes about the next block in the chain, due to either similar mining times or potentially malicious miner activity, two valid chains exist and a chain split occurs. Splits are resolved by majority vote; if a miner receives two valid chains and successfully mines a new block, it will extend one of the chains and other miners in the system will most likely adopt the extended chain and abandon the other. Miners are incentivised to continue mining the longer chain: mining requires effort. If the miner continues to mine a shorter chain, it must perform extra work to "catch up" its chain and potentially miss out on lucrative block rewards if it cannot do so and has to switch later on. However, these mechanics lead to security concerns such as a potential "51% majority attack" [52]. In these attacks, a malicious party controls over half the network's ability to mine new blocks. This gives them the ability to manipulate the mining of new blocks and their contents.

Each block is filled with transactions agents attempt to place onto the blockchain. Agents send these transactions to a node, which then propagates them throughout the network, meaning whichever node mines the next block, all will be able to include this agent's transaction. To prove authenticity, the agent will cryptographically "sign" the transaction with a digital signature from their wallet's private key, a personally identifying cryptographic variable that proves authenticity and ownership of the wallet in question. The signature is a hash that can be quickly confirmed to have come from the private key of the wallet and match the wallet's public key but cannot be used to decrypt the private key in a realistic time.

All blockchains work slightly differently with their own advantages and disadvantages. I will focus on Ethereum as it is the blockchain underpinning my work.

2.1.2 The Ethereum Blockchain

Ethereum [6, 60] was one of the first blockchains ever created. Ethereum is famous for being the first "programmable blockchain". Ethereum is a Turing Complete blockchain that can store and run stateful applications; these applications are treated as "first class citizens", allowing its complex autonomous and variable behaviour not available to its predecessors. In recent years, Ethereum has spawned an explosion of innovation on its platform: ERC-20 tokens built on top of Ethereum and their Initial Coin Offerings (ICOs) [25]; the recent Non-Fungible Token (NFT) mania [58, 57] and Decentralised Applications (DApps) [41]. Ether (ETH) is the core token used to power the Ethereum network. It is intrinsically a store of value and after bitcoin, has the second-largest cryptocurrency market capitalization in the world [12]. It's also used to pay miners and fund computation on the blockchain. ETH has multiple denominations, the smallest of which is Wei: 10^{18} Wei make up 1 ETH. Gwei is another common denomination: 10^9 Gwei make up 1 ETH; Gwei is often used in relation to miner gas fees.

2.1.3 The Ethereum Virtual Machine (EVM)

The EVM is a singular entity, albeit collectively updated and maintained by the distributed Ethereum network, that is often referred to as a form of state machine. Agent and DApp transactions are compiled and executed by the EVM. EVM executions fundamentally transition the overarching blockchain state from one form to another. The EVM is the global store of all active agent wallets and smart contract states. The EVM stores the blockchain's state in a modified Merkle Patricia Trie [16], which acts as a highly compact data store. This trie allows each block to only contain the changes to the state rather than the entire state itself, significantly reducing block size.

2.1.4 Smart Contracts

Smart contracts are the primary mechanism for interacting with the EVM. Smart contracts are written in Solidity: Ethereum's high level, object-orientated programming language [21]. Smart contracts contain global state variables and methods (functions); methods can interact with both global state variables and methods from other smart contracts. They cannot alter state variables from other contracts directly.
A contract lifecycle is as follows:

1. An agent writes a smart contract in the Solidity language.
2. They compile the contract to obtain the bytecode and the Application Binary Interface (ABI). The bytecode must be sent in the deployment transaction. The ABI is the interface between the agent and the EVM, it contains all publicly accessible contract variables and methods that can be called. The ABI is used to translate methods and properties into their corresponding EVM bytecode representations.
3. The agent sends a deployment transaction to the network containing the contract's bytecode. Once the transaction has been accepted, the EVM stores the methods and state of the contract. If there is one, the EVM also executes the constructor method of the contract, then discards it as it would not be of use to store a single-use method.
4. The agent, and others, can then view publicly accessible variables and call non-state altering methods directly. To call a method that updates state or another contract's state altering methods, a transaction must be executed on the blockchain. These interactions are completed using the ABI generated during compilation, linking the EVM representation with a human-readable version.

Smart contracts are first class citizens; they have their own public key and wallet address. Contracts can complete all actions agents can directly with their own wallets, they can even initiate their own transactions.

After deployment, a contract's methods and state variables are immutable outside of the framework of interactive capability the contract was initially deployed with. This comes with its own set of pros and cons: this provides transparency and security for agents interacting with another agent's contract but means a bug in a deployed contract may be extremely serious and difficult to resolve. Today, a new practice for significant, large scale contracts is to deploy them as "upgradable contracts" through the use of a proxy contract. This proxy contract forwards interactions to the main underlying contract. The proxy contract can be pointed to a different contract, allowing the agent to deploy a secondary contract with updated behaviour as and when needed [10].

Table 2.1: Data components required to create a valid Ethereum transaction (TX).

Key	Description
nonce	Wallet nonce, increasing each TX
gasLimit	Max gas units to use
to	Target: agent or contract address
value	ETH, in Wei, to transfer to *to* address
data	function call data, deployments & other messages
sig	ECDSA signature of TX contents for proof of ownership
Before EIP-1559	
gasPrice	ETH fee, in Wei, per unit of gas
After EIP-1559	
maxFeePerGas	Max ETH fee for miner & base fee, in Wei, per unit of gas
maxPriorityFeePerGas	ETH fee for miner, in Wei, per unit of gas

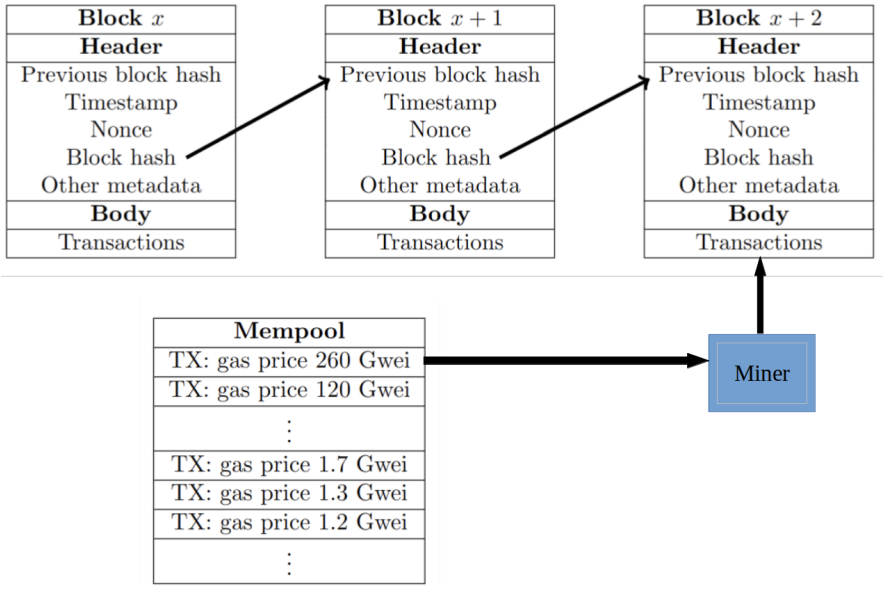

Figure 2.1: A simplistic visualisation of a blockchain and a miner mining a new block with transactions from the mempool.

2.1.5 Wallets, Transactions & Signatures

Agents interact with the blockchain through their wallets. A wallet is made up of a public and a private key. The public key is a hash of the private key and is used to publically identify a user's wallet. The private key is used to authenticate all transactions the agent places through the wallet. Agents can send ETH to other agents or contract wallets and interact directly with smart contracts through transactions.

Ethereum uses the Elliptic Curve Digital Signature Algorithm (ECDSA) [38] to create digital signatures from a hash of the message to be signed, specifically the SECP256k1 curve [59] is used. ECDSA can only produce signatures; it cannot be used for message encryption.

This signature can then be used by a verifier, along with the public key of the wallet to verify the sender has used and is in possession of the wallet's private key, giving it the authority to send transactions.

Each transaction must specify a transaction nonce, starting from 1 at account inception and increasing with every transaction thereafter. The transaction will be dropped by the network if the nonce is invalid. This prevents accidental double transactions for agents and malicious replay attacks.

The full contents of a transaction are outlined in Table 2.1. The full transaction contents are encoded using Recursive Length Prefix (RLP) [24] encoding and sent to a node to broadcast throughout the network if the transaction is deemed valid.

Table 2.2: Example gas units required for different EVM computations and transaction types [17].

Type	Gas Units Required
1 execution cycle	1
add	3
mul	5
balance: get wallet ETH balance	20
call: read-only inter contract method call	20
sload: read from blockchain state	20
sstore: write to blockchain state	100
base transaction: e.g. ETH wallet transfer	21,000
base contract deploy: empty contract	53,000

2.1.6 Blocks

The blockchain is made up of a chain of blocks; miners "mine" new blocks that contain transactions agents have sent by order of gas price, a visualisation of this is depicted in Figure 2.1. With Ethereum, miners secure the ledger and mine transactions using a PoW mechanism which is computationally intensive. Miners all try and solve the same algorithm, once one has done so, they are able to mine the next block; the mined block gets added to the ledger. Each block is mined roughly every 13 seconds. To maintain this timeframe, the mining difficulty is constantly recalibrated, dependent on the global hash power being used to mine blocks.

Computational intensity is measured in Ethereum with "gas". Each block has a "gas limit", currently of size 30 million units, which is the maximum amount of computation allowed in each block. Due to the limited nature of blocks, during periods of high network activity, competition occurs between agents attempting to enter transactions into a pending block. Gas usage of some example computations and transactions are listed in Table 2.2. For deployments and contract methods that change state, these gas usages are substantially higher depending on the complexity of the constructor and the number of state variables being altered or added to the blockchain. Transaction gas requirements for complex deployments and method calls can often exceed 300,000 gas units. Agents have to pay a fee in ETH for every unit of gas they use, set by the current rate of the network; this is set to help control congestion and is expanded upon in Section 2.1.7.

2.1.7 The Mempool & Block Fees

The mempool is a pool of unconfirmed transactions that have been broadcast throughout the network and are waiting to be mined. Agents will set a "gas price" and a "gas limit" for their transaction: the gas price is the cost in ETH per unit of gas the agent provides the miner, the gas limit is the maximum number of gas units the transaction is authorised to use. The gas limit is there to protect both the miner and the agent. The gas limit prevents accidentally high costs through unintended execution loops for the agent and protects miners from arbitrarily complex executions maliciously built to overwhelm them.

When a miner is filling a block with transactions, there are often too many transactions to fit in the block. Miners are self-serving, they choose transactions in an order that provides them with the highest ETH reward. This creates an environment where agents will bid up the gas price during periods of congestion in an attempt to persuade miners to include their transactions over lower-priced offerings.
A transaction lifecycle is as follows:

1. An agent broadcasts a transaction with a specific gas price and limit to a node in the network.
2. The node will analyse the transaction to verify it is well-formed and valid; if the transaction is invalid it will be dropped from the network.
3. The node broadcasts the transaction throughout the network and the transaction is placed into the mempool.
4. A miner successfully mines a new block, it analyses the mempool and chooses transactions by gas price to maximise the reward for mining the block. Once the block is full, the remaining transactions are ignored and will be picked from the mempool in a later block.
5. The transaction is included in the block and recorded onto the blockchain. The transaction will be recorded as "successful" or "reverted". Upon reversion, the blockchain will be rolled back to the

previous state before the transaction was executed; reversions occur when there is an error during EVM execution of the transaction data. Miners still receive the fee for computation even when a transaction is reverted.

6. The difference between the actual gas used and the gas limit authorised is returned to the agent's wallet.

Gas prices are constantly fluctuating. When blocks are consistently less than full due to a low transaction volume, the gas fee required to get a transaction into a block falls also, due to senders not having to compete for a place in a full block.

Even if a block is not full, some agents may desire to be placed "above" other transactions in the block. For example, when multiple agents are competing for and trying to interact with a contract to withdraw the same resource. In these cases, agents may still specify unusually high gas fees, as the order of execution in the block is also determined through this gas price mechanism.

2.1.8 Ethereum tokens

Ethereum is famous for democratising the ability to create new cryptocurrencies and tokens. Tokens are simply smart contracts which provide the functionality of a cryptocurrency: minting, max supply, ownership, transfers among more. The ERC20 token standard [18] is a standard contract frequently used by agents to inherit base standard token functionality that they can use to implement their own tokens. Tokens have been created with this standard amongst others to implement a wide range of use cases including financial assets, rewards, collectables, digital art and more.

2.1.9 Sidechains

Sidechains, or "Layer 2" systems, are independent, Ethereum-compatible systems and blockchains built to run in parallel to the main Ethereum network. Sidechains are built to augment and improve the core functionality of Ethereum in ways that are not possible, too extreme or divisive to be adopted on the main Ethereum chain.

In this project, I initially made extensive use of the Polygon sidechain [55] however, I eventually moved to using the Goerli test blockchain for better MEV-geth support. The speed and cost of activities on the Polygon chain are significantly reduced compared with Ethereum. The underlying blockchains are different, but are intrinsically linked through a Bridge [54], that allows communication and transfer of assets between the two separate ledgers. For example, for a generic ERC-20 token on the Ethereum blockchain that an agent wishes to transfer to Polygon: a smart contract will be created to "lock up" the tokens on the Ethereum blockchain, the bridge and token contracts on the Polygon network will then mint an equal number of tokens on the polygon chain. The process can then be repeated in reverse to send back to the Ethereum chain.

2.1.10 EIP-1559

In August 2021, Ethereum Improvement Proposal 1559 (EIP-1559) [7] was introduced to the main Ethereum network. Previously, the entire gas fee was exclusively received by the miner who mined the block. There was no intrinsic gas fee level built into the protocol which led to confusion over accurate gas fees agents should pay, often leading to wild fee swings & overpayments. Third-party gas fee oracles [22] would calculate approximate current gas fees based on historical network activity.

EIP-1559 updates the Ethereum protocol to split the gas fee into a "base fee" and a "priority fee". The base fee is a fee that is burnt by the network; the priority fee is the "tip" provided to the miner similarly to prior the update. The base fee is constantly updating based on network activity: when the previous block is more than 50% full, the base fee increases and vice versa. This new mechanism was implemented to help agents estimate the gas fees required. The change in mechanism is depicted in Figure 2.2 and the updated transaction contents in Table 2.1.

For simplicity, throughout this project gas price and priority fee are used interchangeably. In this document, gas price should be assumed to be the priority fee post EIP-1559 implementation, base fee is effectively ignored.

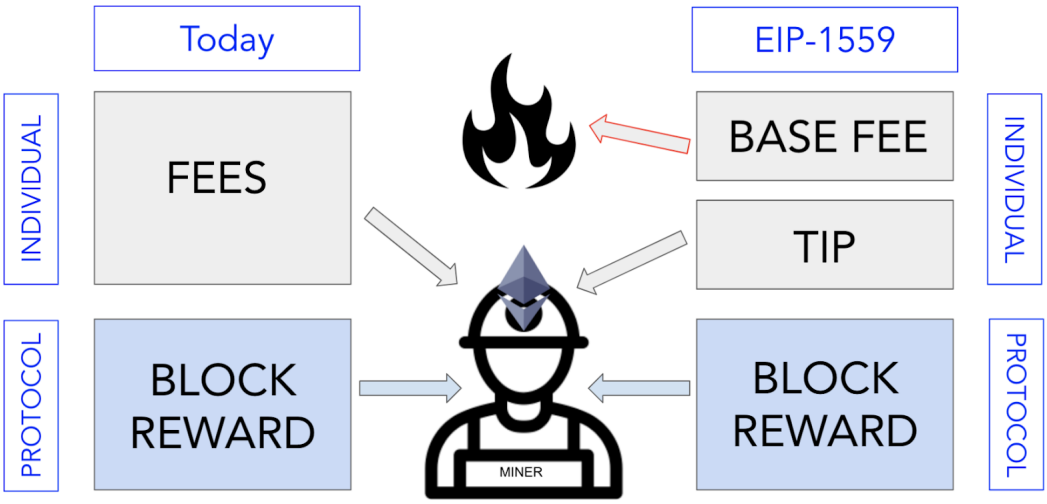

Figure 2.2: A visualisation of the change in gas fee handling by the network after the implementation of EIP-1559 [5].

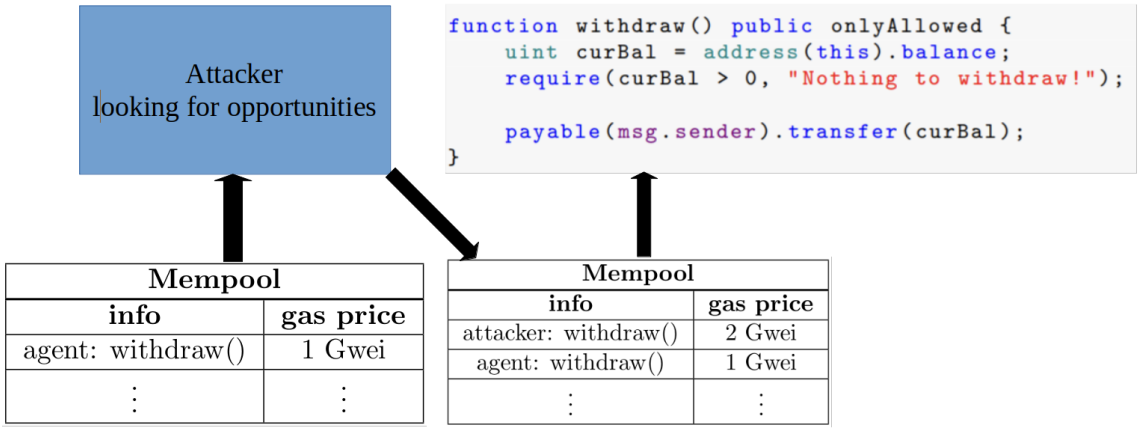

Figure 2.3: An example generalised displacement attack: the attacker is monitoring the mempool for atomically profitable transactions, it identifies a withdraw() TX and sends it's own withdraw() TX with a higher gas price to complete before the original agent's.

2.2 Front-running Attacks in Blockchain

2.2.1 Overview

In most traditional financial markets, unless through some unfair advantage, the competition to place transactions is simply down to the speed at which your transaction request reaches the centralised system running the market, known as first-in-first-out (FIFO) ordering. For a decentralised blockchain such as Ethereum, the transaction ordering and acceptance mechanics add a new layer to ordering competition, giving rise to a wealth of attack vectors malicious parties may try to utilise.

2.2.2 Generalised Displacement Attacks

An agent may be placing a TX that is atomically profitable. In other words, this TX will return the agent more financial value than it costs to place the TX. In this scenario, an attacker is analysing the mempoool of in-flight TXs and realises the agent's TX is atomically profitable. The attacker then executes a copy of this TX themselves before the agent's TX is mined. An attacker will broadcast their TX with a far higher gas price to persuade miners to include their TX before the original agent's. The agent's TX will

then fail and the attacker will reap the reward.

A depiction of such an attack is shown in Figure 2.3. The agent attempts to call a `withdraw()` method of a smart contract, which returns the ETH stored in the contract; the `withdraw()` can also be called by the attacker. The agent sends their TX, the attacker monitors the mempool and simulates the TX in a safe test environment; the TX is shown to provide financial value to the attacker so they send their own TX to the mempool with a higher gas price than the original agent. The attacker's TX is mined first and the attacker receives the reward.

Displacement attacks are known as "generalised" when the attacker has no underlying knowledge of what the TX they are displacing is actually doing, all they require is atomic profitability. An attacker can analyse the transactions currently pending in the mempool, simulate their execution with their own information and determine it is a profitable opportunity. They can then displace the transaction themselves and reap the reward.

2.2.3 Insertion Attacks (Sandwich)

Auto Market Maker (AMM) services such as liquidity pools operate through smart contracts on the blockchain. Liquidity pools contain a balance of two assets, providing the ability to swap between each of them, the cost of one asset rises when the demand for it outweighs the demand for the other asset in the pool. When an agent buys an asset, the asset's price will rise. An attacker can identify a TX such as this and simultaneously front-run and back-run the TX for profit.
In their simplest form, liquidity pools utilise the equation:

$$E * T = K \tag{2.1}$$

E and T are the two asset balances making up the pool, K is a constant initialised with the pool's starting balances. When the amount of asset E increases, asset T must fall to maintain the K constant. The drop in asset T is what is supplied to the agent in return for its asset E.
In the following example Sandwich attack, R is the amount removed from the pool during an asset swap. The example is as follows:

1. A liquidity pool between ETH (E) and TOK (T) tokens exists. The current pool equation is as follows:
$$10(E) * 10(T) = 100(K) \tag{2.2}$$
2. Alice places a TX to swap 1 ETH for TOK. Alice expects to receive 0.91TOK.
3. An attacker identifies this TX, places their own identical buy TX with a higher gas price than Alice's TX. The attacker also places a second TX with a lower gas price than Alice's to sell the TOK they have just bought.
4. The attacker's TX is included before Alice's, the attacker receives 0.91TOK and the pool updates as follows:
$$(10 + 1) * (10 - R) = 100 \tag{2.3}$$
$$R = 10 - \frac{100}{11} \tag{2.4}$$
$$R = \frac{10}{11} \approx 0.91(T) \tag{2.5}$$
$$11(E) * \frac{100}{11}(T) = 100(K) \tag{2.6}$$
5. Alice's TX is eventually filled however, the asset balance in the pool has now changed. Alice only receives 0.76TOK as follows:
$$(11 + 1) * (\frac{100}{11} - R) = 100 \tag{2.7}$$
$$R = \frac{100}{11} - \frac{100}{12} \tag{2.8}$$
$$R = \frac{25}{33} \approx 0.76(T) \tag{2.9}$$
$$12(E) * \frac{25}{3}(T) = 100(K) \tag{2.10}$$

6. The attacker's second TX is filled, swapping the 0.91TOK they received at the more favourable rate as follows:

$$(12 - R) * (\frac{25}{3} + \frac{10}{11}) = 100 \tag{2.11}$$

$$R = 12 - \frac{100}{\frac{25}{3} + \frac{10}{11}} \tag{2.12}$$

$$R = \frac{72}{61} \approx 1.18(E) \tag{2.13}$$

$$\frac{660}{61}(E) * \frac{305}{33}(T) = 100(K) \tag{2.14}$$

7. The attacker receives 1.18ETH in return and has made 0.18ETH profit from the Sandwich attack. Alice received 0.15 less TOK than she intended.

2.2.4 Suppression

Attackers may desire to prevent an agent from placing a TX. For example, an agent may be attempting to include a TX that sells a large quantity of assets, if the agen succeeds, the asset price would fall significantly. The attacker prevents the agent from selling his assets for a period of time until the attacker has sold his own holdings and is no longer affected by the sale. Or, deadline-based smart contracts such as lotteries: the last TX before the deadline wins, or the winner is chosen from the entries to the lottery. The attacker sends his own lottery entries and then attempts to prevent further entries from other agents being included until the end of the lottery.

Attackers can prevent an agent's TX from being mined by flooding the network with TXs with a higher gas price, filling up the available block space. If the attacker's TXs are all of higher gas price and there is a large enough quantity of collective computation to reach the block gas limit, the agent's TX will not be included until these circumstances change. This is also known as "block stuffing".

2.2.5 Sybil Attacks

Some opportunities are intended to be limited in some way per agent. DApps will implement protections into their smart contracts to limit agent's abilities to interact multiple times. The contract could record the addresses of agents and prevent them completing a second interaction. Or, the contract could limit interactions per wallet address with a cooldown period between interactions. For example, a lottery could be implemented to allow one entry per wallet, or an asset sale could limit buys to one every 5 seconds.

A Sybil attack occurs when an attacker spawns multiple temporary wallets to provide the illusion of multiple independent agents, effectively nullifying the protections outlined above.

Ethereum TX fees are relatively high: at today's rates, the average TX fee is $10.84USD [9]. These high costs somewhat prohibit a proportion of Sybil attack opportunities. However, on side chains such as Polygon, transaction fees are much lower, often fractions of a single cent and these attacks are much more attractive and feasible.

2.2.6 Malicious Miners

Miners have complete control over transaction ordering in a block. Miners have already been shown to be acting in their own best interest and to maximise their own block reward. In most cases, this gives rise to the transactions being ordered by gas price, as this maximises the reward to the miner. However, if a miner could re-order the transactions or replace a transaction with one of their own and extract more value, there is strong reason to believe they will do so; this is a serious concern.

A time bandit attack is an MEV extraction technique that occurs when a miner attempts to "rewrite history" by mining competing blocks to those that have just been mined to take advantage of past MEV opportunities. A miner will identify an opportunity mined in the previous block, mine a competing block at the same height and attempt to advance their own chain, leaving the original block as an "uncle" block. Whilst difficult to do, requiring between 40-51% of the network hash-rate, the ability and incentive for miners to execute them have been outlined in the *Flash Boys 2.0* [14] paper.

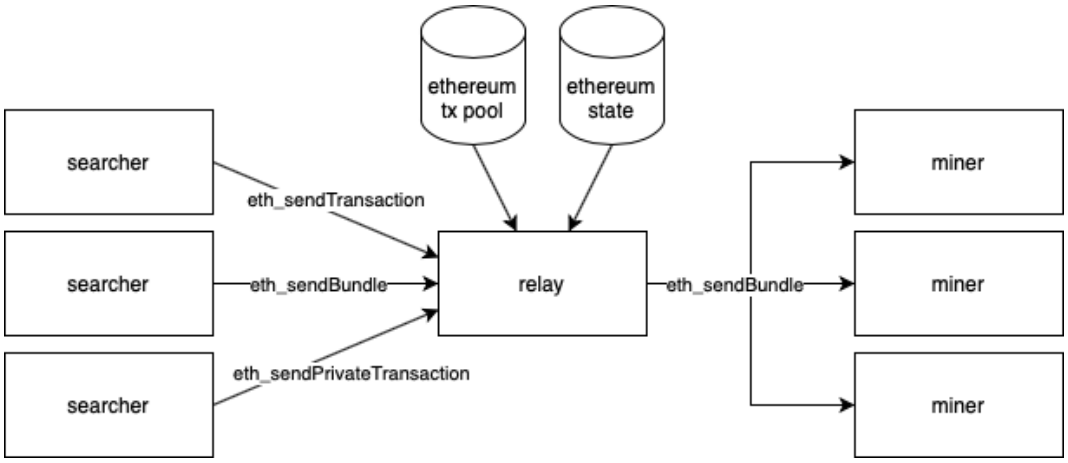

Figure 2.4: An overview of the transaction flow when using Flashbots Auction/MEV-geth. Sourced from [29].

2.3 Maximal Extractable Value (MEV)

MEV was first defined in *Flash Boys 2.0* [14] as "miner extractable value", which is the deterministic value that can be extracted by miners from placing and re-ordering transactions in the network through methods outlined in Section 2.2. However, assuming miners are honest, this is also available to all other agents in the system and therefore has since been generalised to "maximal extractable value". At some point in 2022, Ethereum will switch to PoS: TXs will be included in the blockchain via "validators" rather than miners in what's known as "The Merge" [20], further rendering the term "miner extractable value" inapplicable.

2.3.1 Flashbots Auction / MEV-geth

Flashbots [29] is a research group that was formed to address some of the issues surrounding MEV. Flashbots introduced the new concept of a "Flashbots auction'; these auctions provide a mechanism for agents to send TXs and have them included by miners without ever being visible in the public mempool. The TXs are routed directly to miners and are provided with a reward for including the directly routed TX. Miners will choose to include these TXs if the reward for doing so is greater than if they were to include TXs from the traditional mempool exclusively.
The flashbots auction is implemented as follows:

1. Agents send their TXs using a new RPC method *eth_sendBundle*. *eth_sendBundle* also allows agents to bundle multiple TXs into a single call.

2. "Relayer" clients act as an intermediary and receive the bundle; if the bundle is valid and well-formed, it will forward the transactions to miners who have implemented the updated protocol.

3. A miner running an augmented version of the *go-ethereum* miner client [19] known as "MEV-geth" [27] receives the TX bundle. If including the bundle would benefit the miner more greatly than the next TX in the mempool, it will include the TXs from the bundle into its block.

Using this mechanism, reverted TXs are not included on the blockchain, saving the agent wasted fees on failed TXs and reducing congestion on the block. This is one of the reasons for including an intermediary relayer: agents do not need to pay for invalid bids, potentially opening up the network to spamming of invalid bundles which miners cannot currently mitigate. The updated TX flow is outlined in Figure 2.4.

2.3.2 Effect on Front-running Attacks

Flashbots auctions are a major benefit for agents attempting to mitigate front-running risk; TXs are no longer visible before being included in blocks which prevents an attacker from identifying an opportunity before it's too late. However, this does not prevent potential attacks from malicious miners such as time bandit attacks mentioned in Section 2.2.6; miners still have complete control over TX inclusion and order.

At the time of writing, flashbots estimates that over 85.5% of the network's hash-power comes from miners running MEV-geth. Well-known DApps such as 1inch [1] and MistX [44] have already migrated to the protocol to provide their agents greater protection.

Flashbots has produced a web dashboard to show insights into the prevalence of MEV extraction today at https://explore.flashbots.net/. At the time of writing, in the last 30 days, over $7 million of MEV has been extracted by miners and agents. In the last 30 days, according to flashbots, of the $7 million extracted, 99.378% was from arbitrage exploitations and 0.622% from asset liquidations. However, it's important to note that flashbots only analyses purely decentralised extractions from specific DeFi protocols and is not exhaustive as outlined here [26]; the prevalence and types of MEV extraction could be far greater.

MEV also includes non-front-running type attacks; these attacks do not require knowledge of the mempool to execute as they can be determined from the current blockchain state. The Flashbots auction does not prevent these attacks and can actually help attackers execute them: the protocol provides an attacker with a way to execute an attack discreetly, without other attackers detecting their attempt and then trying to displace them; the free cost of reversion also means MEV opportunities which are no longer available do not cost the attacker in fees for their TXs failing.

2.4 Related Work

2.4.1 MEV Prevalence & Front-running Attacks in Blockchain

In 2019, *Flash Boys 2.0* [14] brought to light the prevalence of front-running and generalised bots on the Ethereum blockchain and its effects on network security; they were also the first to introduce term "Miner Extractable Value (MEV)", later replaced with maximal extractable value. This paper introduces proxy contracts as a means for attackers to operate multiple TXs in a single atomic operation, to allow reversion on any point of failure. They introduce atomic arbitrage opportunities in DeFi and how attackers compete for these opportunities through PGAs. They deployed nodes on the Ethereum network with with an augmented version of the node software go-ethereum to record unmined and replaced TXs in the mempool in addition to the final included TXs usually stored in the blockchain state. At the time, they observed significant occurrences of PGAs in practice, and estimated 1.6 million ($) was availble in pure abitrage revenue profits per year, from asset exchanges they supported. This paper also presents a formal model of PGAs, concluding the most optimal strategy for attackers is to "blind raise", meaning they continuously increase their bids, without waiting to react to other attackers; this allows an attacker to bid with a reduced latency than waiting to see other attacker's actions. Further, they conclude attackers should and do increase their bids by the minimum increment allowed, otherwise incentivising the collective attackers to unneccessarily raise the bid, reducing the profit potential of the opportunity. The paper then looks into MEV from the perspective of the miners; they conclude MEV opportunities create significant security risks to the fundamentals of blockchain security, incentivising miners to participate in time-bandit attacks and arbitrary re-ordering of TXs in mined blocks.

December 2021, a paper [47] looked further into quantifying MEV extraction and formalised a generalised front-running algorithm. This paper utilised the Blockchain Extractable Value (BEV) term in place of MEV. This paper puts forwared that a " a rational miner with a 10% hashrate will fork Ethereum if a BEV opportunity exceeds 4× the block reward." Emphasising the security concerns this brings to blockchain systems overall. They estimate that over the prior 32 months, 540.54 million ($) had been extracted by attacker on the Ethereum blockchain specifically from sandwich attacks, liquidations and arbitrage opportunties they were able to measure. This paper, to its knowledge, was also the first to provide a concrete algorithm for a generalised displacement algorithm. This algorithm is a generalised displacement algorithm that replaces the sender address with the attacker's address only, they do not look into TX parameter alterations. This paper also looks into the rising prevelance of private relays for sending TXs, which MEV-geth is one of; they conclude these relayers to do not reduce network congestion and increase the chance of network attacks elicited by miners.

March 2021, a paper [61] looks into the analysis of real-time block state for the extraction of MEV, rather than the analysis of in-flight transactions. The paper focuses on arbitrage and more complex MEV opportunities that can be identified from the blockchain state. They use negative cycle detection algorithms like the Bellman-Ford-Moore algorithm to identify available arbitrage opportunities. They produce a "DeFiPoser-SMT" MEV identification algorithm that is shown to identify the 0.46 million ($) MEV opportunity that was available for 69 days and was extracted during the bZx attack [11].

2.4.2 Front-running Mitigations

December 2021, the paper [4] looks into three categories of front-running mitigations: fair ordering, batching of blinded inputs and private user balances and inputs; the former preventing arbitrary transaction re-ordering, the latter two hiding the contents of the TXs and the user's intent. Fair ordering would require protocol level changes such as "*y-batch-order-fairness*", where TX's are ordered based on when $y\%$ of nodes receives each TX; this is shown to introduce new TX front-runners that attempt send a TX and meet the required threshold before the original TX can do so. During batching, users' TXs would be combined and sent through some intermediary smart contract, only releasing user balances once all TXs have been completed. Secret user input stores can also be maintained by DApps off-chain to prevent visibility until the completion of the operation however, this significantly reduces the permissionless nature of the DApp.

March 2022 this year, a paper [35] looks into mitigations available to common front-running attacks in blockchain and their corresponding pros and cons, specifically focusing on mitigating transaction re-ordering. The paper outlined common front-running attacks such as Sandwich attacks and generalised Displacement attacks in blockchain systems. The paper puts forward numerous DApp specific mitigations, such as an AMM automatically extracting any MEV created by an agent's TX and giving it to the agent themselves, or by redesigning Decentralised Crypto Exchanges (DEXs) entirely to enforce fair ordering with off-chain implementations. They propose a trusted centralised third party or an algorithmic committee as a fair ordering enforcer, requiring significant protocol change. A model where a TX is guaranteed to only succeed if the blockchain state has not changed after submission, mitigating Sandwich attacks but does nothing for Displacement attacks, as these do not require the agent's TX to succeed. All methods put forward require significant Ethereum protocol change or specific smart contract implementations.

At a similar time, the same authors released a second paper [34] looking into the specific vulnerabilities production AMMs still face from Sandwich attackers despite current DApp specific implementations. They show that despite slippage tolerance parameters set on liquidity pools, specific parameters must be set for each individual TX to prevent Sandwich vulnerabilities; they put forward an optimal slippage tolerance algorithm to mitigate most Sandwich attacks that they encourage AMMs to adopt.

Chapter 3

Project Execution

My project execution steps can be categorised into three overarching groups:

1. Highly performant, asynchronous blockchain interaction software: the system needed to analyse the mempool; send, rebroadcast and cancel transactions; manage wallets and deploy and interact with contracts. Transaction sending with MEV-geth must also be implemented to provide the agent front-run mitigation.

2. Implementation of attacks scenarios: smart contract creation, deployment and interactions with the attacker and agent to simulate different front-running attacks. The ability for agents to both use the traditional transaction sending method and the MEV-geth method, which should prevent the attacker from succeeding, are implemented.

3. Dashboard creation: a web-based dashboard where experiments established in group 2 can be tested using real wallets on a test network (Goerli) by interested parties. The dashboard must allow users to configure the agent to both send transactions traditionally and with MEV-geth, implementing and validating the mitigation provided thereby. Users should have visibility into the transactions being placed by both the agent and attacker; the outcome of the experiment must also be clear.

3.1 Base Blockchain Interaction System

3.1.1 General Setup

Concurrent Execution Setup

Software interacting with blockchains require lots of IO-bound network requests. These requests take time and leave the script idle in a synchronous setting. Asynchronous environments allow the software to perform other activities whilst waiting for IO activity and return to it once available, improving the system's performance. Additionally, blockchain transactions are not atomic: they require sending and then monitoring in case of rebroadcast necessity and information retrieval once complete. For this reason, I wanted a transaction monitor (Section 3.1.4) to consistently run concurrently with the primary processing of the scripts.

By default, code written in the Python language is synchronous. The Asyncio module is used to provide concurrent capabilities to Python software through the use of coroutines: threads of logic that can be run concurrently, switching between each-other when threads are waiting for IO operations. The main logic and background worker threads are run concurrently as individual coroutines. `thread_wrapper(func)` and `indefinite_worker_wrapper(func)` decorators wrap the main execution thread and the background worker threads respectively. These decorators both read from a global variable `PROGRAM_ACTIVE` and allow the workers to run indefinitely whilst the main thread is active. In case of an error in any thread, the whole system would shut down as the `PROGRAM_ACTIVE` variable is set to `False`. An `async_runner(wrapped_funcs : list)` function is used to run the wrapped coroutines in an Asyncio event loop. These utilities can be found in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/asyncio_utils.py.

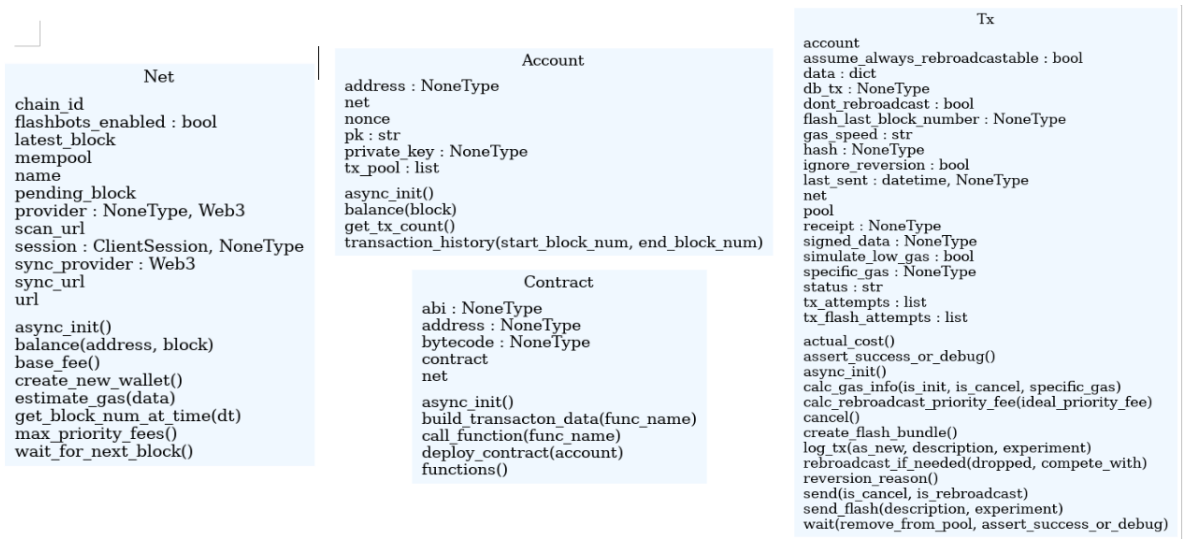

Figure 3.1: UML diagram of the core blockchain classes, properties and methods implemented. Visualised with Pyreverse, part of Pylint [40].

3.1.2 Network Management

The base of the system is `Net: class`, used to abstract the underlying Web3.py module interactions and generalise the connection to any network or node endpoint. The object is initialised with the information of the network node being connected to, for example either a local Ganache blockchain node or a Goerli test-net Infura node. This object can then be used by the rest of the system to interact with the underlying node connection. This class also implements methods to analyse the network and state of the blockchain; for example, it can be used to retrieve the balance of any wallet, retrieve the current network base gas fee and an estimate of the current priority fees required among more.

All properties and methods are shown in Figure 3.1. The class can be found in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/net/net.py.

3.1.3 Account Management

Private Key Security & Storage

The security of private keys are of extreme importance to agents: if an attacker were to access an agent's private key, they would have complete control over their account and would have the ability to steal all assets. For this reason, I opted to never store raw account keys to the filesystem. Instead, I encrypt the account keys with a password and store this key store in an environment file. Each time the program is executed, the password to decrypt the keys is entered. The only unencrypted keys exist in the program's running memory, maximising security.

To encrypt the keys, I use the eth-account [30] `LocalAccount.encrypt()` method that utilises AES-128-CTR [46] and then save the result to an environment file. These encrypted keys can then be decrypted later.

Account Wrapper

The `Account: class` is used to manage an agent's wallet. The wrapper is passed the wallet's key information, either directly with the address and private key, or it is passed an encrypted version of the keys and the passphrase it then uses to decrypt them.

When sending TX's, the nonce provided must specifically match the total number of TXs the particular wallet has previously sent. This is initially calculated through a node request with `account.get_tx_count()` but maintained internally thereafter, being manually increased every time a TX is sent; this is done to prevent further unneccessary and slow requests. However, this means the nonce value will become out of sync if the agent sends TXs through another piece of software in parallel to running this software; this should currently be avoided.

The `account.transaction_history(start_block_num, end_block_num)` must be used to identify potential historical balance changes during visualisation in Section 3.3.4. This functionality is not available through

Web3.py: the Etherscan API was directly used via the requests module to obtain this information. The `account.balance(block)` can then be used to retrieve balance information at a particular historical block.

Properties and methods are shown in Figure 3.1. The class can be found in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/account/account.py.

3.1.4 Transaction Management

Transaction Class

The `Tx: class` is used to create, send and manage each individual transaction. During instantiation, the account object is passed in, along with the configuration of the transaction data. Optionally, gas price and a specific gas limit can be specified. For testing purposes, the TX can be configured to provide too little gas, never rebroadcast, continuously rebroadcast and allow sending of TXs guaranteed to revert.

The agent can use the `tx.send()` method to sign the TX data with the account's private key and send the signed TX; the account's nonce will also be incremented. The `tx.wait()` method can then be used to wait for the outcome of the transaction. Due to the TX monitor, concretely waiting for a tx to complete is entirely optional.

Generic configuration such as the current TX nonce, gas usage estimation and chain id are set by this object. Gas priorities/speeds of `GAS_SLOW, GAS_AVERAGE, GAS_FAST` can be specified during instanstiation, defaulting to `GAS_AVERAGE`; the network will be polled for the 1st, 50th and 90th percentile of recent historical gas prices to assign the applicable gas speed using the `tx.calc_gas_info()` method.

An in-flight TX can be cancelled using the `tx.cancel()` method; a dummy TX is sent with the same nonce as the in-flight TX with a higher gas price. The miner will then mine this TX instead of the original one due to the higher gas price. This method is not guaranteed to succeed; the miner may have already mined the in-flight TX or does so before acknowledging the cancellation TX.

Properties and methods are shown in Figure 3.1. The class can be found in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/tx/tx.py.

Transaction Monitor

Upon sending a TX, its status is marked as `TX_PENDING`. The TX then needs to be continuously monitored; if the state of the network changes and the average gas fees increase to significantly higher than the initial values set for the configured gas speed, the miners are likely to drop the TX. The miners will include other TXs with higher gas fees in place of the agent's TX; the transaction must be rebroadcast with higher gas fees.

The `tx_monitor()` function runs as an `indefinite_worker` continuously polling the currently pending TXs. This thread accesses the TXs through a `TX_POOL: list` global variable; when a TX is sent, it gets added to this pool and removed by the monitor upon completion. The monitor eventually sets all TX statuses to one of `TX_SUCCESS, TX_REVERTED, TX_DROPPED`.

The monitor will continuously poll the network for the TX receipt using the TX hash initially returned when the TX was sent and acknowledged. If a receipt is found, the TX has been included into a block; the receipt is saved to the TX object and its status updated to `TX_SUCCESS`. The TX hash is also queried, if the node cannot find the hash in the mempool for at least ten seconds, the TX has been dropped and is rebroadcast. If the TX hash is found by the node but the network average gas fees have significantly increased, the TX will be rebroadcast using the `tx.rebroadcast_if_needed()` method.

The transaction monitor function can be viewed in Appendix C.

MEV-geth Transactions

Web3.py does not support the RPC endpoints implemented by MEV-geth miners, specifically *eth_sendBundle*. Flashbots provides a Web3.py integration module that is injected into the Web3.py instance to implement these methods. The `tx.send_flash()` method utilises this endpoint to send the TX through this protocol. TXs must be sent in a bundle of TXs with minimum gas of 42000. Given a simple transaction is only 21000 gas, if the gas is too low, a dummy transaction is also included to meet this minimum requirement. MEV-geth transactions are synchronous; they must be sent every block until they are successfully accepted by a miner. This method attempts to send the transaction in ten consecutive blocks until it is accepted or raises an error. Pseudocode for this method is provided in Algorithm 3.1.

MEV-geth is supported on the main Ethereum Network, the Goerli test-net and the Polygon sidechain main network. I intended to run experiments on the Polygon sidechain however, in practice, it seems the

```
bundle = list(txData)
if txData["gas"] < 42000 then
    bundle = list(txData, createDummyTx())
end
blockNum = getCurrentBlock()
for x=blockNum; x+1; x < blockNum + 10 do
    success = sendFlash(bundle, blockNum + x)
    if success = True then
        break
    end
end
```

Algorithm 3.1: Sending a TX using MEV-geth / Flashbots Auction. The bundle must have a minimum of 42000 gas, a dummy TX will be included if the target TX does not meet this requirement. The bundle is sent and specifies the target block; the logic will attempt to include the TX in ten consecutive blocks until one succeeds. Written in pseudocode.

fraction of the network supporting MEV-geth is too low: my TXs were seldom accepted within the ten block limit, meaning no miner running MEV-geth had mined a block during that period. For this reason, I decided to use the Goerli network for all of my MEV-geth interactions.

3.1.5 Contract Management

The `Contract: class` is used to create and interact with Solidity smart contracts deployed on the blockchain. To interact with an existing contract, the contract's ABI and address are entered. To interact and deploy a new contract, the compiled Solidity bytecode of the contract must be entered in place of the contract address.

`contract.call_function()` can be used to interact with non-state altering contract methods found in the ABI to retrieve their return values. Since the state of the blockchain does not need altering, no TX is needed; the return value can be calculated from querying the current blockchain state.

When calling methods that update blockchain state, a TX is needed. The function name and parameters are encoded into transaction data with the `contract.build_transaction_data()` method; this data can be used to instanstiate a `Tx: class` object which can then be sent and awaited.

To deploy a contract, the `contract.deploy_contract()` method can then be used. The contract's information and constructor method are encoded into TX data and are sent as a TX object. Once the TX has succeeded, the contract's new address is saved to the contract object.

Properties and methods are shown in Figure 3.1. The class can be found in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/contract/contract.py.

3.1.6 Mempool Visibility

The `Net.mempool()` method is used to view currently pending transactions across the network. This information comes from the node provider, the underlying RPC endpoint being requested is *txpool_content*. The node provides the hash of the TXs, along with all of their TX data.

During local testing with Ganache, this method worked perfectly. However, after much research, I came to realise this endpoint is not available with Infura nodes; Infura themselves state they are working to add support [37] however, the issue is ongoing and cannot it cannot be assumed to be resolved in the near future. To get access to this endpoint, another node provider would have been needed that did provide access to this RPC endpoint, or I would have had to run my own full network node.

Time constraints led me to replace this functionality with a workaround. The global `TX_POOL` variable, used by the transaction monitor, contains all of the pending TXs sent by this software. Given I am using this system for an experimental implementation of agents and attackers running on the same software, it provides the functionality needed for a representation of the mempool during these experiments. This limitation is explored further in Section 4.3.1.

3.1.7 Simulating Transactions

I developed a method using the local Ganache blockchain to simulate transactions. Ganache can be used to fork from real networks. Forking the network essentially means a duplicate of the network: the forked node has access to the main networks state and any TXs sent on the fork update the forked state but do not affect the original network.

Ganache is spun up as a Python subprocess and killed after the simulation. Furthermore, I also implemented the system so that Ganache can fork from itself, using a different network port. This meant I could run these simulations even if I were already using the Ganache blockchain. These forks can be used to simulate transactions on the current blockchain state and analyse the updated state. The fork can then be killed as it is no longer necessary. Using the Ganache CLI, Ganache can fork from itself as shown in Listing 3.1. This functionality is meant to be used to fork from other real networks, but this is a valuable side effect for local simulations.

A `run_ganache()` function was implemented to run the Ganache instance as a subprocess. This method returns a secondary function that kills the underlying Ganache subprocess once called; this should be used when temporarily forking for simulations, or when running tests with different Ganache configurations to prevent hanging Ganache instances. Parameters to the function can configure the Ganache blockchain in different ways: if forking, the node provider url and block number to fork from can be configured; the time taken to mine a block, chain id and port can also be configured. Ganache also provides the ability to "unlock" wallets; this means that you can send TXs from real wallets, albeit in a test environment, from a real network without having to sign the TX with a private key. This is a useful tool to test smart contracts that only allow access from accounts you do not have access to.

The function can be found in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/utils.py.

```
#!/bin/bash

# Simple local ganache blockchain accessible at http://127.0.0.1:8545:
./node_modules/.bin/ganache --port="8545"

# A forked version of the original blockchain accessible at http://127.0.0.1:8546:
./node_modules/.bin/ganache --port="8546" --fork.url="http://127.0.0.1:8545"
```

Listing 3.1: A bash script to start a local Ganache blockchain instance, then fork from that instance with a secondary Ganache instance to run risk-free simulations on. This shows how Ganache can fork itself during testing, when using a real network, it would be forking from the node provider's URL instead and only one instance of Ganache would be running.

3.1.8 Test Environment

To ensure the functionality of the system, I created tests to validate all parts of the blockchain interaction software. The Python Unittest module [50] was used to create a test suite. For each test, a Ganache blockchain instance was spun up as a subprocess using the `run_ganache()` function with the required configuration and killed after each test's completion. The temporary Ganache instance was setup with account wallets, funded with 1000ETH, that could be used to test functionality in a safe environment.

For speed, Ganache can be run with an "instamine" configuration. With this setting, TXs are instantly mined when they are received, rather than waiting for a specic interval between mining blocks, increasing execution speed. Some tests require periodic block mining to test functionality; when testing cancellations and rebroadcasting, a longer block time was needed to allow the software time to send out replacement TXs. For these tests, block times between were set between 0.5 to 5 seconds.

TXs were thoroughly tested. A single TX, multiple TXs in the same block, rebroadcasting TXs and cancelling TXs were all verified to function as expected. Deployment of contracts, call and transacting contract interactions, reversion of TXs and more were also verified. The experiments outlined in Section 3.2 were also included in the tests. The experiment functions were run and the expected balance changes of both the attacker and agent were verified.

All tests created are outlined in Figure 3.2. The utilities used run Ganache, setup and tear down tests, is available in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/test_utils.py.

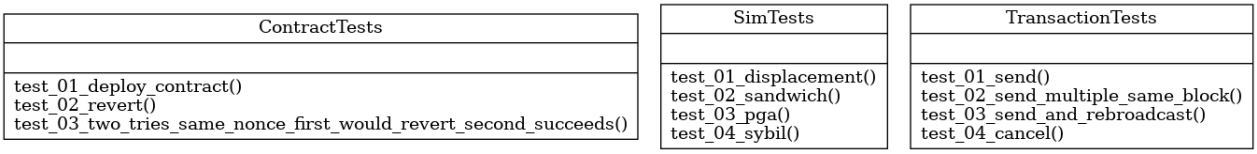

Figure 3.2: UML diagram of blockchain tests and experiments created using the Unittest Python module. Visualised with Pyreverse, part of Pylint [40].

3.2 Attack Implementations

The source code for all of the experiments outlined in this section can be found in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/primary/sims.py.

3.2.1 Generalised Displacement Attack

$forkedNet$ = forkNet(net)
for tx *in mempool* **do**
  $attacker$.net, tx.net = $forkedNet$
  $startBal$ = $attacker$.balance()
  tx.data[“from”] = $attacker$.address
  tx.send()
  $finalBal$ = $attacker$.balance()
  if $finalBal > startBal$ **then**
    $attacker$.net, tx.net = net
    tx.gas_speed = GAS_FAST
    tx.send()
  end
end

Algorithm 3.2: A simplified generalised Displacement attack algorithm written in pseudocode. For each TX in the mempool, the attacker replaces the TX “from” address with their own, simulates the TX on a Ganache fork and if profitable, displaces the original TX by sending a replacement with a higher gas price.

```
1   def convert_params_to_attacker(params, attacker):
2       new_params = []  # Converts param names dict to just param values list
            also
3       for param_name in params:
4         if Web3.isAddress(params[param_name]):
5             new_params.append(attacker.address)  # Replace with attacker address
6         else:  # Leave param untouched if not an address
7             new_params.append(params[param_name])
8     return new_params
9
10  func_name, params = contract.contract.decode_function_input(tx.data["data"])
11  updated_params = convert_params_to_attacker(params, attacker)
```

Listing 3.2: Logic to replace the address information in the parameters of a smart contract call with that of an attacker. Requires access to the smart contract's ABI.

As outlined in Section 2.2.2, a Displacement attack occurs when an attacker completes a similar TX before the original agent can complete theirs. To implement this attack, I configured the attacker to fork the network with a Ganache local instance, for each TX in the mempool he would replace the identifying information with his own, simulate the TX and if profitable, displace the original TX with a higher gas price. Pseudocode for such an attack is outlined in Algorithm 3.2. For brevity, only the TX “from” address is translated to the attacker's.

The contract parameters that contain addresses are also translated to the attacker and is shown in Listing 3.2; this currently requires the contract being interacted with to be known in advance: decoding TX function and parameter information requires availability of the smart contract's ABI. In this scenario the smart contract is known, mitigating the ABI issue. However, with unknown contracts, parameter alteration would not be possible. This limitation is explored further in Section 4.3.2.

A simple holder contract was created for this experiment: a naive contract that can receive ETH; the ETH can then be extracted by anyone. The contract can be funded with the `receiveFunds()` method, funds can then be withdrawn with the `withdraw()` method. In practice, as we don't want to be front-run ourselves, these methods are limited to the agent and attacker accounts. If an account other than the agent or attacker attempts to interact with the contract methods, their TX would revert as they are not authenticated. The contract is included in Appendix B.1.

The contract is initially funded with 0.1ETH using the `receiveFunds()` method. After, the attack is implemented as follows:

1. Agent attempts to extract the ETH using the `withdraw()` method with a gas speed of `GAS_AVERAGE`.
2. The attacker monitors the mempool for TXs. For each TX, the attacker replaces the agent's address information with their own, forks Ganache and sends the TX on the forked chain, i.e. with no risk. The attacker sees he's profited on the fork; the attacker then sends the same TX on the original blockchain with a gas speed `GAS_FAST`.
3. The attacker's TX is included first, extracting the 0.1ETH and ending with the status `TX_SUCCESS`.
4. The agent's TX is included afterwards, there is no ETH left in the holder contract and the TX ends with the status `TX_REVERTED`.

Without having any underlying knowledge of what the attacker is doing, he is able to profit from this scenario and leave the agent with nothing. Real TXs executed can be found in Table 3.1.

Table 3.1: A real Displacement attack on the Goerli network created during an experiment. The `withdraw()` method of the contract provides the caller with the assets in the contract, if there are no assets in the contract the call will revert.

Step	Gas Price	Status	Explorer Link
Attacker: withdraw()	2.92 Gwei	TX_SUCCESS	https://goerli.etherscan.io/tx/0x74a37735380c61df2b55c4c6e62fb057b8568098c99d660c5171a73f4973b760
Agent: withdraw()	1.25 Gwei	TX_REVERTED	https://goerli.etherscan.io/tx/0xe9bd6f56fea0835e1e3c5d8f4dfeeffe652a66738c28abef96107ddacfb0540c

3.2.2 Sandwich Attack

```
for tx in mempool do
    if tx.data["to"] = simplePool.address then
        if simplePool.getFuncName(tx) = "swapEthForTokens" then
            tx.data["gasPrice"] = GAS_FAST
            tx.data["from"] = attacker.address
            tx.send()
            sellTx = simplePool.createTx("swapTokensForEth", attacker, GAS_SLOW)
            sellTx.send()
        end
    end
end
```

Algorithm 3.3: A simplified Sandwich attack algorithm written in pseudocode. For each TX in the mempool, if the attacker recognises the TX is interacting with a known liquidity pool and is a swap, it front-runs the TX with its own information with a higher gas price, then back-runs the TX with a reverse swap with a lower gas price.

As outlined in Section 2.2.3, a Sandwich attack occurs when an attacker simultaneously front and back-runs an agent's swap TX in a liquidity pool or equivelent. To create an environment for a Sandwich attack to occur, I implemented a simple liquidity pool smart contract; the contract is listed in full in Appendix B.2.

The contract includes an internal token, managed through the `POOL_TOKEN_BALANCE`, `AGENT_TOKEN_BALANCE`, `ATTACKER_TOKEN_BALANCE` contract properties. The pool tokens are abstract and are never sent out to an

agent; this was done for simplicity and removing the need to implement a token specification such as ERC20. The ETH contract balance (`address(this).balance`) and TOK contract balance (`POOL_TOKEN_BALANCE`) make up the equation as follows:

$$\text{ETH} \times \text{TOK} = \text{K\_VAL} \tag{3.1}$$

`K_VAL` is initialised after funding the pool with ETH using the `receiveFunds()` method. The `swapEthForTokens()` method can then be used to swap the ETH sent in the TX for tokens using the aforementioned equation; this swap occurs by increasing the balance of the agent/attacker with the contract property `AGENT_TOKEN_BALANCE/ATTACKER_TOKEN_BALANCE`. `swapTokensForEth()` can then by used to do a reverse swap, swapping all the tokens the user holds back for ETH at the current exchange rate. Both of the swap methods implement the equations outlined in Section 2.2.3.

To implement this attack, I configured the attacker to monitor the mempool for `swapEthForTokens()` TX calls; upon finding an applicable TX, the attacker will send their own identical `swapEthForTokens()` with a higher gas price than the agent's TX, then send a `swapTokensForEth()` reversal TX with a lower gas price than the agent's TX. The two attacking TX's will mine before and after the agent's TX, thus profiting from the increased asset price the agent's TX causes. Simplified pseudocode of the attack logic can be found in Algorithm 3.3.

Initially, 0.5ETH is funded to the contract using the `receiveFunds()` method; this automatically sets up an exchange rate between ETH (E) and the internal TOK token (T). By design, Solidity uses Wei as the denomination, i.e. $0.5\text{ETH} = 5 \times 10^{17}\text{Wei}$. The following equation (in Wei) is setup after initialisation:

$$(5 \times 10^{17})E \times (5 \times 10^{17})T = 2.5 \times 10^{35}(K) \tag{3.2}$$

The attack then runs as follows:

1. Agent sends 0.05ETH with the `swapEthForTokens()` method with a gas speed of `GAS_AVERAGE`.
2. The attacker is monitoring the mempool for buy transactions directed to known pools (in this case, this simplistic pool only).
3. The attacker recognises the agent's TX and initiates its own identical TX to `swapEthForTokens()` with a gas speed of `GAS_FAST`. It initiates a TX to `swapTokensForEth()` with a gas speed of `GAS_SLOW`.
4. The attacker's first TX completes first, the agent's second and the attacker's second TX completes last. The agent receives less TOK than expected and the attacker profits ETH.

This experiment was built to work both locally with Ganache for testing and in production with the Goerli network. Real TXs executed can be found in Table 3.2.

Table 3.2: A real Sandwich attack on the Goerli network created during an experiment. The attacker is shown to both front and back-run the agent, utilising higher and lower gas prices to extract profit from the agent.

Step	**Gas Price**	**ETH Change**	**Explorer Link**
Attacker: swapEthForTokens()	3.22 Gwei	−0.05ETH	https://goerli.etherscan.io/tx/0xa17a1ccc2e6d3dee9a20587d62ad56430e29de6a51518bd750b60024405e0a6a
Agent: swapEthForTokens()	1.78 Gwei	−0.05ETH	https://goerli.etherscan.io/tx/0xdccdf8da341bb16cf329ec746659d9164885286b1178eaac014c864f4aaf9305
Attacker: swapTokensForEth()	1.25 Gwei	+0.066ETH	https://goerli.etherscan.io/tx/0x909f2b881726570010c0c3c0aa016b768a1015f22874040715980bc3949a17b8
Agent: swapTokensForEth() NB: for completeness and done separately	1.77 Gwei	+0.034ETH	https://goerli.etherscan.io/tx/0x7603095cd7ff886aa520b1aa994e9990fdafc0d2d31533bf3efaad558ffdfeea

3.2.3 Priority Gas Auction (PGA) Competition

In this scenario, an attacker and an agent are both aware of one another, actively bidding (rebroadcasting) against each other to complete their transaction first. Actors incrementally outbid one another until the block is eventually mined and the last bid wins. This also occurs when two attackers compete for a Displacement attack opportunity, for example. The same holder contract is used as in Section 3.2.1.

When rebroadcasting, networks require the TX gas price to increase by a minimum percentage. There is no set standard percentage required across all nodes but is usually between 10-20%; for guaranteed acceptance I have used 30%. The `tx.rebroadcast_if_needed()` method is provided access to the competing TX, increasing the gas price by the minimum percentage and adding 30% on top of the competing TX; this accounts for potential in-progress price increases in the competing TX.

The execution of this implementation leads to an exponential gas fee increase provided to the miner. I only have a limited supply of Goerli ETH available to me; to prevent wasting it, this experiment was exclusively run using the Ganache local blockchain. With ommissions, a PGA executed locally is outlined in Table 3.3.

These auctions are extremely detrimental to the losing party. Given their reverted TX is still included, they must pay the greatly inflated gas fees to the miner.

Table 3.3: A real Priority Gas Auction on a local Ganache blockchain instance created during an experiment. Two agents are shown to compete with incrementally higher gas prices until the block is eventually mined.

Step	**Gas Price**	**Status**
Attacker: withdraw()	2193.62 Gwei	TX_SUCCESS
Agent: withdraw()	1523.1 Gwei	TX_REVERTED
Attacker: withdraw()	1085.31 Gwei	TX_DROPPED
Agent: withdraw()	752.77 Gwei	TX_DROPPED
⋮	⋮	⋮
Attacker: withdraw()	1.88 Gwei	TX_DROPPED
Agent: withdraw()	1.37 Gwei	TX_DROPPED
Attacker: withdraw()	1 Gwei	TX_DROPPED
Agent: withdraw()	1 Gwei	TX_DROPPED

3.2.4 Sybil Attacks

Whilst no experiment environment has been created for a Sybil attack scenario, the asynchronous implementation of the software has made their execution straightforward. A `Net.create_new_wallet()` method has been implemented to create an arbitrary number of wallets on the network. Given the asynchronous nature of the software and the underlying transaction monitor, these wallets can be first funded concurrently, then execute their target TXs concurrently. An implementation using the software is provided in Listing 3.3 for an arbitrary target TX.

```
async def sim_sybil(base_agent, tx_data, number_of_agents, fund_val):
    agents = []

    # Spawn the agents:
    for x in range(number_of_agents):
        wallet_info = base_agent.net.create_new_wallet()
        agents.append(
            await create_account(
                base_agent.net,
                address=wallet_info["address"],
                private_key=wallet_info["private_key"],
            )
        )

    # Fund the new wallets:
    txs = []
    for agent in agents:
        txs.append(await create_tx(base_agent, {"to": agent.address, "value":
            fund_val}))
```

```
    await asyncio.wait([tx.send() for tx in txs])
    await asyncio.wait([tx.wait() for tx in txs])

    # Execute the distributed attack:
    txs = []
    for agent in agents:
        tx_data_personal = copy.deepcopy(tx_data)
        tx_data_personal["from"] = agent.address
        txs.append(await create_tx(agent, tx_data_personal))
    await asyncio.wait([tx.send() for tx in txs])
    await asyncio.wait([tx.wait() for tx in txs])
```

Listing 3.3: An implementation of a Sybil attack using the software. The controlling agent passes in the TX data, number of spawned wallets and amount of ETH to send to each spawn; the script spawns the wallets, funds them and then executes the attack concurrently.

3.3 Experiment Dashboard

The dashboard is available at http://3.66.150.226. If the site goes down for any reason, a site overview of all sections can be found on YouTube at https://youtu.be/3SqUuUnZFlE.

This dashboard was created using a pre-existing web template framework created by myself, Zachary Stucke, **previously before this project**. This template implements management scripts, a React [42] front-end with generic components, global store and request handlers, utilities and admin interface, a Django [49] backend with generic database structures, API handlers and utilities. It also implements a task management system using Celery [8], allowing scripts to run outside the usual web request-response cycle.

This template is licenced software; for this reason, it has been ommited from the public project repository at https://github.com/zakstucke/ethereum-front-running. Therefore, recreating the frontend dashboard and web portions of the project from the public repository would require significant engineering effort.

API views, database models, frontend React pages and logic that have been created during this project's lifetime are still available in the repository in the "backend" and "frontend" folders respectively.

3.3.1 API Calls & Initiating Experiments

API calls are sent from the frontend to both update the frontend visualisations and for users to initiate experiments on the backend. Three API calls were implemented:

1. **GetTxs:** an automatic, periodic call that returns the most recent 30 transactions completed by the system.
2. **RunSimulation:** a user initiated request to initiate a new experiment.
3. **GetBalances:** an automatic API call that returns balance information for the agent and attacker over the last hour that can be used to create a line graph on the frontend. This API call can be manually re-fired to update for any new information.

The logic implementing these API calls can be found in the public repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/primary/views.py.

Each of the experiments take between 30 seconds and 2 minutes to complete; when a user initiates an experiment, this length of time is too long to complete during the response of an API call sent by the frontend. For this reason, the experiments are initiated as Celery background tasks: they continue running after the API call has completed. These tasks run asynchronously and update the experiment state continuously.

These tasks can be found in the project repository at https://github.com/zakstucke/ethereum-front-running/blob/main/backend/primary/tasks.py.

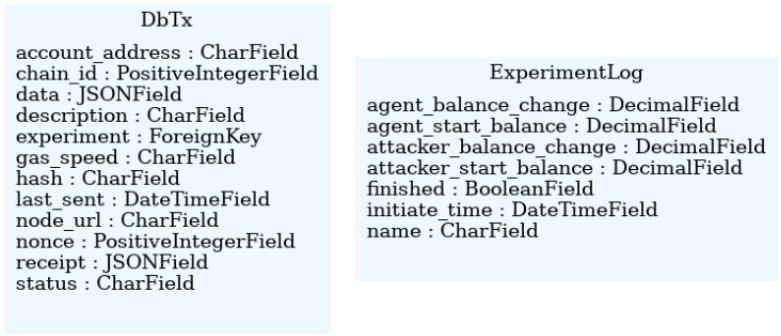

Figure 3.3: UML diagram of the database model classes and fields. Visualised with Pyreverse, part of Pylint [40].

3.3.2 Experiment Configuration Form

As shown in Figure 3.4, a configuration form is rendered to allow site users to configure the experiment. Displacement, Sandwich and PGAs are available. Users can also specify the execution type as "Traditional", sending TXs using the traditional method with visibility pre-completion in the mempool, or using MEV-geth, preventing pre-completion visibility and preventing the attacker from interfering.

PGAs fundamentally use the traditional execution type; a PGA requires visibility into the mempool for the agents to bid against eachother. For this reason, if the PGA experiment is requested with the MEV-geth execution type, the form will reject the request. For simplicity, the system enforces a single experiment to be running at any one time. If an experiment is already running, this form will reject the request and inform the user to wait for a few minutes.

3.3.3 TX History

Experiment & TX Database Models

In order to maintain a log of experiments and TXs, database models for both were created. When sending to the frontend, the database object fields are serialized into JSON format.

An ExperimentLog model was created containing the experiment's name, initial account balances, start time and a boolean field to mark an experiment as finished or not. This was used to provide information relating to the current in-progress experiment and to prevent further experiments from being initiated whilst one was ongoing.

A DbTx model was created to record a TX and its state throughout its lifecycle. A DbTx instance of a TX could be created through the `tx.log_tx()` method. The DbTx model is also assigned to an ExperimentLog model through the use of a foreign key field. The transaction information recorded in the DbTx model is a subset of information from the main Tx class that is useful to visualise on the frontend. These TX classes were intentionally kept separate as database updates are inherently slow, they require significant IO when updating the database; given the blockchain interactivity software is built to be as fast as possible, these updates should only happen at specific intervals when required.

A complete list of database model fields can be found in Figure 3.3.

TX Log Viewer

As shown in Figure 3.5, visibility into the TX history is provided on the dashboard through a list of recent TXs. The list is refreshed every three seconds with the most recent TXs in the database; by default, the list is configured to show only the most recent experiment TXs but this can be changed to see up to 30 recent TXs across experiments. Incomplete TXs can also be filtered out of the list view.

When showing for a specific experiment, the experiment configuration is included at the base of the list; upon completion, the agent and attacker outcome balances are also shown at the top of the list.

Initially, each TX shows its description, current status, priority fee and send time. Upon clicking a particular TX, the complete information provided by the DbTx class is provided in a pop-out modal, including the TX data and a link to the TX on the explorer. This is only available when the TX is on the Goerli network and the TX has been completed. The TX popout is shown in figure 3.6

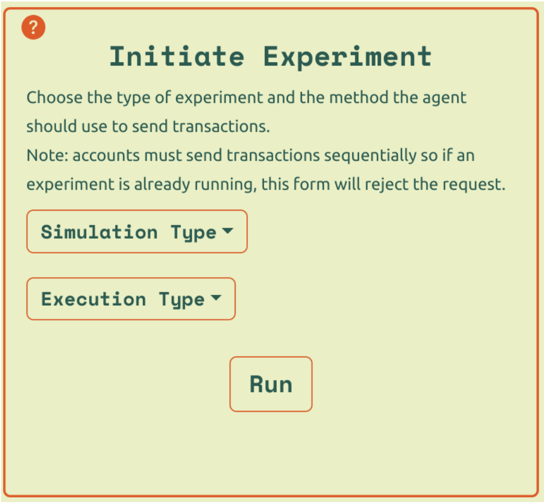

Figure 3.4: Dashboard: the experiment configuration form allowing users to configure and initiate experiments themselves.

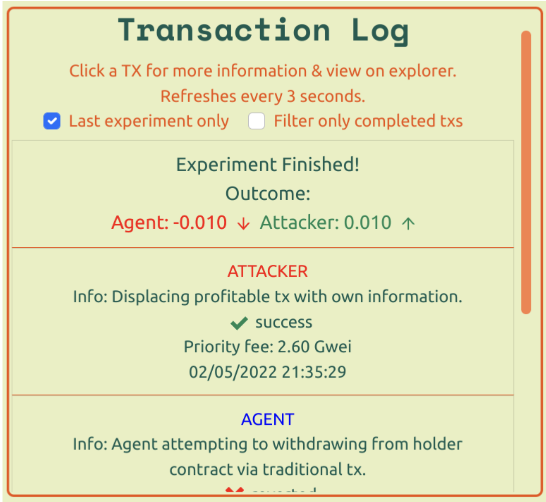

Figure 3.5: Dashboard: the transaction log/history view providing insight into ongoing experiments in real-time and historic experiments.

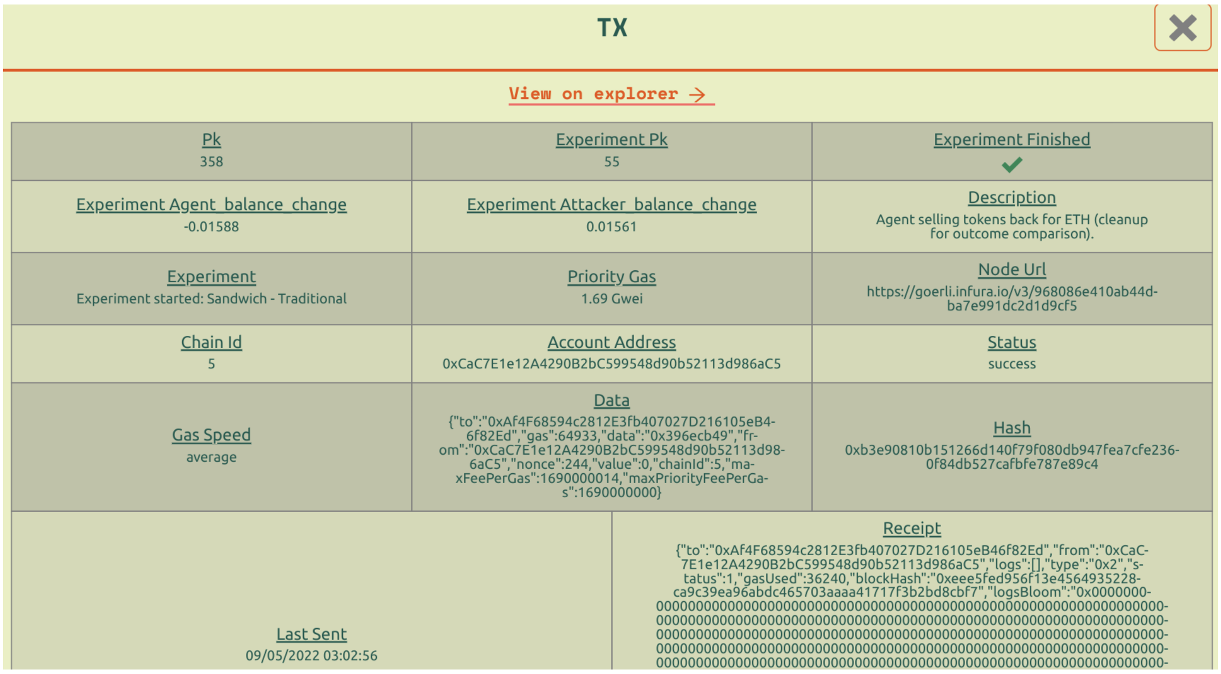

Figure 3.6: Dashboard: the popout modal shown after clicking on a TX in the TX log. The full TX information is shown along with a link to the TX on the blockchain explorer, providing the local Ganache blockchain was not used.

3.3.4 Balance Monitor

As shown in Figure 3.7, the balance changes of both the attacker and agent wallets are visualised with a line graph. Transactions for each account are polled over the last hour. Balances are then calculated for the block height of each transaction; the balances at the start and end of the period are also shown. These balances can be refreshed using a button below the graph.

Querying past TXs is not supported natively by nodes; the Etherscan API was used to complete this calculation. A call to `https://api-goerli.etherscan.io/api?module=block&action=getblocknobytime&timestamp=TIMESTAMP_HOUR_PREVIOUSLY&closest=before&apikey=API_KEY` was used to get the closest block number to an hour ago; this block number, along with the current block were then passed to `https://api-goerli.etherscan.io/api?module=account&action=txlist&address=ADDRESS&startblock=START_BLOCK&endblock=CURRENT_BLOCK&sort=asc&apikey=API_KEY` to retrieve a list of TXs

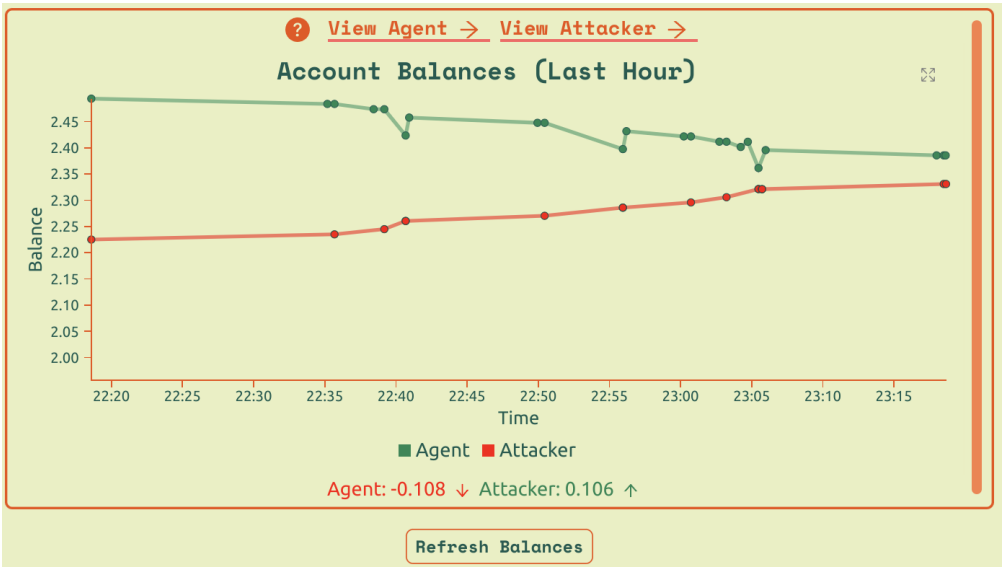

Figure 3.7: Dashboard: a visualisation of the changing agent and attacker balances on the Goerli test-net. These are the two accounts used during Displacement and Sandwich experiments on the web dashboard.

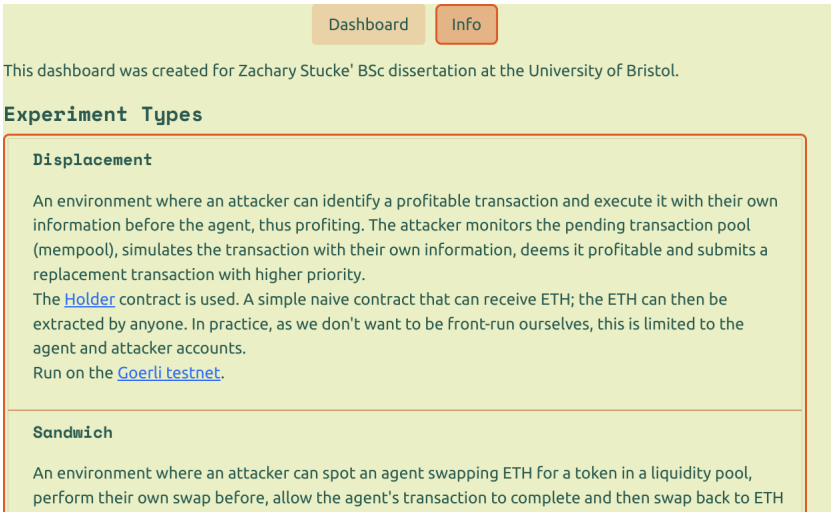

Figure 3.8: Dashboard: a portion of the secondary information tab containing further information related to the experiments. The two smart contracts utilised during the experiments are also listed.

for the specified account over the time period. Balances were then queried using the usual node endpoint at the specific blocks referenced in each TX to obtain datapoints for all changes in balances for both accounts. This data is then used to plot a line graph on the frontend dashboard depicting the balance changes over the period.

3.3.5 Info Page

The site page contains two tabs, the main dashboard tab it defaultst to, and also an information tab, titled “Info”. The info tab provides further information relating to the experiment and execution types. Both smart contracts used during the experiments are also listed on this page. A portion of this tab is depicted in Figure 3.8.

3.3.6 Project Deployment

This project needed to be accessible from the internet however, the project’s value is not in its web implementation but the underlying blockchain software implementation. For this reason, I chose a deployment process that was easy to implement, rather than being much more rigourous and building a

production-ready website.

I setup an AWS Elastic Compute Cloud (EC2) instance of size t3.small in the Frankfurt region. I attached an Elastic IP to this instance, guaranteeing a stable public IP address (3.66.150.226). This instance is attached to an 8GB storage volume; the database requirements for this project are low, therefore I am running the project's PostgreSQL [33] database directly from this same instance inside a Docker [36] container. This container uses persistent storage to prevent data loss in case of malfunction. I imported the project contents to this EC2 instance and ran the webserver on port 80, providing full network access to this port in the security settings.

If this were a production build I would have done a few things differently. I would have implemented the database as an Amazon Relational Database Service (RDS) instance. I would have configured a domain name for the site and setup HTTPS access for security. I would have also run the Celery background tasks on a different instance. However, given this is not the value of the project, I deemed my current implementation sufficient.

3.4 Dashboard Presentation

On 6th May 2022, I presented the dashboard and an introduction to this work to the Financial Technology with Data Science MSc cohort of the University of Bristol. This presentation was recorded and can be found on Youtube at `https://youtu.be/AQ_8G7B1Q7Q`.
The presentation briefly covered the following points:

1. A blockchain overview.
2. How miners choose in-flight TXs from the mempool to include in their next mined block based off of the gas fees the TX provides to the miner.
3. Generalised Displacement attacks and how they occur.
4. AMMs, specifically liquidity pools and the basic implementation equation.
5. Sandwich attacks and how they occur.
6. PGAs and why they occur between different attackers going for the same opportunity.
7. MEV-geth and how it provides some protection to agents from these types of attacks.
8. An introduction and tutorial on how to use the dashboard created in this project at `http://3.66.150.226`, providing implementations of these attacks.

This presentation and dashboard were well received by the group and may be used as a teaching resource in the future. A review of the presentation written by Theodoros Constantinides, a Teaching Associate present, is listed in Appendix A; a feedback form was also given to the students. Students' responses and subsequent analysis are outlined in Section 4.5.

Chapter 4

Critical Evaluation

4.1 Simulation Time

For each block, Ganache only has to be forked once: the simulation accounts have an arbitrary amount of ETH available so a reset of the simulation environment is not needed between simulations. Through analysis, the time taken to spin up Ganache on my local machine averages to 1.06 seconds; the average simulation then takes 0.49 seconds. Ethereum blocks are mined approximately every 13 seconds.

Between the initialisation of the fork and each simulation, only 28 TXs could be simulated from the mempool each block. In practice, even less time would be available as a period near the end of the block would have to be reserved for actually sending the displacement TXs.

In terms of gas units, the maximum block size of an Ethereum block is 30 million gas units. In the worst case scenario, where every TX uses the minimum gas possible (21000), the maximum number of TXs in a block is as follows:

$$\lfloor 30000000 \div 21000 \rfloor = 1428 \text{ TXs} \tag{4.1}$$

I attempted to calculate the time needed between the sending of a TX and mining of a block to guarantee the miner's acknowledgement of the TX; I calculated the time from initiating the `Tx.send()` method, to the availability of the tx hash from the node provider. The resulting time was approximately 0.2 seconds however, whilst the node provider may have received the TX, further time will be taken to propagate to all available miners, which I have not been able to accurately measure.

To continue implementing a worst case scenario, I will assume TXs must be sent two seconds before a block is mined for the networks collective acknowledgement of the TX. Therefore, 9.94 seconds remain for TX simulations after ganache is started and the end of the window to send TXs. The simulation time available per TX is as follows:

$$9.94 \div 1428 \approx 0.00696 \text{ seconds} \tag{4.2}$$

Assuming every TX in the mempool is unseen, in the worst case scenario, the current simulation implementation would need optimising by approximately 7000%. Clearly, more thought on the implementation of the simulation techniques would be required for anything more than an experimental system.

4.2 Other Attackers

In these experiments, the contracts were specifically set up to revert on any interactions from other agents; this was done to prevent the attacker from being front-run themselves, as the contracts are both inherently vulnerable by design.

In real-world environments attackers are not executing in a closed ecosystem. Attackers will be competing for opportunities through PGAs. As shown in the PGA experiments, PGAs increase the fee sent to miners significantly. PGAs can often reach a point where further bids will cost more than the value of the opportunity, meaning the vast majority of the reward is given to the miner.

An example of this is shown in the PGA experiment in Table 3.3. The final gas price was 2193.62 Gwei. The holder contact `withdraw()` method was being called and gas used was 28,208 units equaling a fee of 0.0619ETH. If the opportunity were smaller than this, the attacker would have lost money. Furthermore, the agent whose TX reverted had a final gas price of 1523.1 Gwei, leading to a definitive loss of 0.0430ETH.

Attackers could use MEV-geth themselves to get around PGAs as other attackers will have no visibility into others' bids. This creates a sealed-bid auction environment where attackers will attempt to guess what others will bid and attempt to outbid them, placing only a singular bid each. The added advantage to attackers using MEV-geth is that they can operate through intermediary smart contracts that will revert on an unprofitable outcome; given MEV-geth does not include reverted TXs, this protects the attacker from the potential loss from missing an opportunity but still being included in the block.

4.3 Limitations of Approach

4.3.1 Lack of Mempool Visibility

As mentioned in Section 3.1.6, the RPC endpoint *txpool_content* is not available when using Infura nodes. I only discovered this when migrating the experiments from my local Ganache blockchain instance to the Goerli test-net, where I utilise the Infura node for connectivity. Time constraints limited me to solving this with a quick fix - I updated the `Net.mempool()` method to return information from the global variable `TX_POOL` instead of the real information pulled from the node.

Given the agent and attacker are both operating from the same system in these experiments, this approach provides a working solution. However, if this were a real environment, this approach would not work as the local pool would have no visibility into other agents' TXs. This is a severe limitation of the current implementation.

To remedy this, another node provider that implements the RPC endpoint *txpool_content* would be required, or a full local node instance would need to be run to access the endpoint directly.

4.3.2 Contracts Method Parameter Alterations

In the case of the generalised Displacement attacks in Section 3.2.1 using the aforementioned holder contract, the attacker can alter the TX with their own information by simply altering the "from" data parameter with their own address. In this scenario, this is all that's needed to make a profitable TX for the attacker. However, it may often be the case where parameters to the contract method are addresses or other parameter information which needs altering to the attacker's information. Contract parameter information is encoded using the contract's ABI; if the ABI is not known, it is not possible to decode the parameter inputs and re-encode with the attacker's.

For the Sandwich attacks in Section 3.2.2, the attacker is dealing with known contracts that have been manually included. In this scenario, it's possible to decode input parameters as the ABI is available. I implement the logic in a `convert_params_to_attacker()` function that takes the contract's ABI and uses it to parse the input params and convert any address parameters to the attackers; this functionality could be extended to test multiple different combinations of updated address parameters and untouched parameters, as some address fields may need to be left untouched to reach a profitable TX.

For many professional contracts, the creators behind them upload a verified source and ABI for the smart contract to Etherscan; the Etherscan API has a method for retrieving these ABIs programmatically [23]. This API call could be used to build up an internal database of ABIs for different contract addresses; the attacker could then use this when an ABI is available. Many smart contracts never get their ABIs' uploaded and, at present, cannot be decoded by the attacker.

4.3.3 Simulating From Start of Block

For all experiments in Chapter 3, when the agent sends the vulnerable TX, the attacker analyses the mempool and executes an attack, the logic waits for the start of a new block to initialise. This was done to guarantee the success of the experiment for test and visualisation purposes: the attacker would have time to analyse the TX, deem an opportunity to be available and execute the opportunity successfully before the agent's TX is mined.

In the real-world, agents' TXs will be placed at random times during the mining of a block. Some TXs will mine before the attacker has noticed there is an opportunity. More dangerously, some will mine after an attacker has initialised their attack. The attacker's TXs are still included in the block, leading to a loss for the attacker.

The attacker could rectify this issue by operating through both an intermediary smart contract and utilizing MEV-geth. The attacker could create a contract that executes the attacking TXs but reverts upon an unprofitable outcome. Given reversions are not included when using MEV-geth, the attacker

would utilise MEV-geth to send the TX to the intermediary contract. The TX would not be included if the attack would fail at the point of inclusion, effectively reducing the risk for the attacker to zero.

4.3.4 Ganache for Simulations

As mentioned in Section 4.1, using Ganache to fork the blockchain and simulate transactions is relatively slow. If attempting to run an attacker in a production system, it would need to be simulating thousands of TXs per block, unavailable with the current setup without significant computational power and multithreading.

Ganache provides a useful and easy to use initial method to show proof of concept and allows small players to enter the system. However, it's clear to simulate the number of TXs required per block would require running a dedicated local node and setting up an environment to run transactions locally with an invalid state for a real production system.

4.3.5 Ganache for PGA Experiment

For the PGA experiment in Section 3.2.3, a Ganache local blockchain was used. Goerli ETH is test Ether however, in recent years has become harder and harder to get a hold of. The usual way of receiving test Ether is through Ethereum faucets: sites which provide a small quantity of Ether to developers. When I started this project, there was a shortage of Goerli ETH and no faucets were operating; I was only able to obtain 5 ETH after requesting it on a Reddit forum [51]. Recently, a new faucet has been created that offers 0.05 Goerli Eth every 24 hours [3], which I have been using to top up my supplies.

During the PGA experiments, due to the bidding process, the miner received approximately 1-2 ETH in gas fees between the agent and attacker. Given this, if I were to use the Goerli network for this experiment, I would have used up my supplies within five experiments and it proved unviable.

The downside of using a local Ganache chain for this is that time-delays from operating in a decentralised system are ignored. The information sharing between the agent and attacker is close to synchronous and would be far more sporadic if it were to be done on a real network, providing additional insight. I could have worked around this by providing a hard limit on the maximum bid of agents in the PGA to cap the fees given to the miner but still analyse the process of the PGA.

4.3.6 Scoped to mempool based TXs only

This project has been exclusively focused on in-progress, pending TXs visible in the mempool. It does not look into the dynamics of a PGA type interaction between two agents both using MEV-geth, creating a sealed-bid auction, nor does it look into attack styles that analyse the current state of the blockchain and previously mine TXs to identify current opportunities.

Other attacks include types such as arbitrage between pools and DApps, but also other attacks directed at individual agents. For example, if an agent were to identify the holder contract in Section 3.2.1 as insecure, they could then monitor for `receiveFunds()` TXs, and instantly realise a profitable `withdraw()` TX is available without having to monitor the mempool at all. An attacker could build a system to test contracts on the blockchain automatically to find these vulnerabilities and then monitor them. Mempool based attackers would not be able to compete in these scenarios as they have no TXs to analyse (assuming the original attacker sends their TX with MEV-geth).

4.3.7 Liquidity Pool Sandwich Protections

Liquidity pools were surmised with the $E \times T = K$ equation in Section 2.2.3. Whilst this provides a basic implementation, it's a gross simplification of production pools. Significant upgrades to the algorithm design would be needed to work against production AMMs. Uniswap [56] is the largest AMM on the Ethereum network. Whilst previous iterations of the pool used the aforementioned equation, its current v3 version [2] augments its implementation to utilise concentrated liquidy at specific price intervals.

Uniswap implements a 0.3% fee per swap which is given to liquidity providers. This fee limits Sandwich attack availability to reasonably large opportunities as the attack must profit at minimum 0.6% plus miner fees. Furthermore, agents can set price slippage tolerances on their swaps currently defaulting at 0.5%; if a Sandwich attacker swaps first and the asset price changes by more than the configured tolerance, the agent's TX will revert, saving them from the attack.

Whilst these mitigations do provide protections to naive Sandwich attackers, as shown in this paper [34], complete protection is only provided based on variable, calculated slippage tolerances which AMMs do not provide by default to agents.

4.3.8 MEV-geth & Malicious Miners

As introduced in Section 2.2.6, miners can be malicious themselves. It is assumed that TXs will be picked from the mempool by order of gas price, however, miners have the ability to re-order and replace TXs with their own in order to extract MEV themselves. MEV-geth provides no protection from miners, it simply protects agents from other attackers viewing the mempool; the miner who mines the block still has complete access. Whilst still providing significant protection from other agents in the system, the requirement to trust the miner is still a significant limitation in the current mitigations provided by MEV-geth.

As of writing, the Flashbots organisation is upgrading its MEV-geth protocol continuously. MEV-geth v0.6 [28] is currently in its alpha. v0.6 introduces a new RPC method *eth_sendPrivateRawTransaction* which does not get broadcast to anyone other than the target miner, reducing the visibility of the TX pre-inclusion even further. Flashbots aim is to eventually privatise the contents of a TX even to the miner; if this is eventually successful, this limitation would have been rectified.

4.4 Completeness

4.4.1 As an attacker

As an attacker, a real implementation of both a generalised Displacement algorithm and Sandwich attack algorithm have been produced on a real blockchain. The only section that would have to be altered to make the algorithms work in practice, with at least some effectiveness, is to properly implement the mempool viewing either using a different node provider or running a local node.

Whilst not essential, the contract decoding without ABI knowledge, or at worst by retrieving from Etherscan, would significantly increase the effectiveness of the Displacement algorithm. It would allow the attacker to also front-run TXs where the address information is encoded into the input parameters.

Whilst also not essential for a "working" attacker, faster simulation using a local node or other improvements over Ganache simulations would also be desired. Whilst in its current form the system could analyse TXs and front-run them, it would not be able to analyse a significant quantity of unmined TXs before the block was mined and the pending TXs were included.

4.4.2 As a Defending Agent

As an agent, this work has implemented an environment where the usage of MEV-geth is proven to protect them from being front-run by another non-miner agent, thanks to the lack of pre-inclusion visibility into the mempool.

As outlined in Section 4.3.8, MEV-geth still puts agents at the risk of malicious miners, albeit at a much smaller risk in comparison to the entire decentralised system.

Therefore, through lack of better existing alternatives, this project concludes that all agents who have to complete atomically profitable TXs should utilise MEV-geth as shown here to provide significant security improvements over traditional TXs.

4.4.3 As an Educational & Informative Tool

The dashboard created at `http://3.66.150.226/` provides the ability for users to configure Displacement attacks, Sandwich attacks and PGAs, using both traditional and MEV-geth TX types. This dashboard provides the following potential learnings as-is to its users:

1. Users can learn what generalised Displacement attacks, Sandwich attacks and PGAs are and what type of TXs they send that could be vulnerable to them.

2. Users involved in creating contracts can view and understand the two smart contracts implemented to create these attacks. These users can understand the common vulnerabilities they should avoid when creating smart contracts to prevent these attacks from being available.

3. Users can learn about and test out MEV-geth to understand how it protects them from other attackers who may be attempting to front-run them.

4.5 Presentation Feedback

I received feedback from the presentation (available at https://youtu.be/AQ_8G7B1Q7Q) outlined in Section 3.4 through a review attached in Appendix A and a feedback form given to the presentation audience; the form results are outlined in Table 4.1. The review was carried out by Theodoros Constantinides, a Teaching Associate in Financial Technology.
Among more, Theodoros states:

> *"The session was really well planned and delivered. Zak demonstrated an in-depth understanding of the topic, and in a short time managed to introduce the audience to the key ideas and problems of MEV and front running attacks. Moreover, Zak's dashboard provides an excellent resource for teaching such topics, as it provides a way to visually show the attacks taking place in real time using a test-net and provides short descriptions explaining how the attacks operate. Although the topics of Zak's presentation are not covered directly in our unit, they are important for our students and his dashboard could be useful for our teaching in the future."*

Theodoros outlines the quality of the dashboard, the insights it brings to his students and how it could be used as a teaching resource in the future, despite the fact the topics are not directly covered in the Financial Technology with Data Science MSc unit. Only four responses were received from the feedback form; as this is a relatively small set of results, no definitive conclusion can be drawn. However, it is useful to see some general insights into the audience's reactions.

Response 1 is positive, they feel the dashboard provided insights for all areas questioned. They do state that more animations could be provided for greater insight. Whilst I loosely agree with this feedback, I think animations to existing content provided would distract the user from the underlying processes they are trying to understand. Rather, I would agree that an additional animated real-time mempool visualisation would be a great addition to the dashboard; such an addition would provide more insights on top of the TX log and help with viewing the competition between the agent and attacker in real-time.

Response 2 is particularly negative, they feel no insights were drawn from the dashboard. However, their free-form feedback states "*The colour scheme applied within the dashboard could be different. Specifically, it makes the data hard to grasp.*" which implies this was the reason for the lack of insight. In light mode, the dashboard's primary colours are dark green, light green and orange (specifically #125b50, #e9efc0 and #ee5007 hex colour codes). Deuteranomaly [13] is the most common form of colour-blindness affecting 6.45% of the population; this condition limits the sufferer's ability to distinguish between red and green colour shades. Given its high prevalence in the population, this could have been affecting the author of response 2. To maximise the availability for all, when designing a colour palette for a website, conditions such as these should be taken into account; in this case, the colour scheme could be improved to prevent such issues.

Response 3 is positive for Displacement attacks, Sandwich attacks and PGAs but neutral for how MEV-geth helps mitigate them. In this scenario, I would assume the user did not attempt to run an experiment with the MEV-geth transaction type to see the agent succeed in their activities. However, this should be emphasised in the information portions of the dashboard that MEV-geth hides pending TXs from attackers; as it stands, this is only inferred through running an experiment and could be emphasised more strongly.

Response 4 was positive for Displacement attacks, PGAs and MEV-geth but neutral for Sandwich attacks. Their free-form feedback states: "*Didn't really understand the sandwich attacks, could have gone through an example.*" This is valuable feedback as the user seems not to grasp the workings of either a liquidity pool or a Sandwich attack. It would be helpful to add examples of each attack to the information tab to make sure the user is able to go through and fully understand what is happening if they choose to.

Overall the feedback is generally positive. The dashboard does seem to provide value to most respondents. A key insight from the feedback is to make the site's design more accessible and easy to view. Another key insight is to provide more general information into the workings of the experiments and the significance of MEV-geth. Between Theodoros' review and the feedback received, validation as a useful teaching resource in its current state has been achieved.

Table 4.1: Feedback collected from a questionnaire [43] after the presentation outlined in Section 3.4. Created with Microsoft Forms.

	This dashboard has helped me understand how generalised Displacement attacks work in blockchain.	This dashboard has helped me understand how Sandwich attacks work in blockchain.	This dashboard has helped me understand how Priority Gas Auctions work in blockchain.	This dashboard has helped me understand how an agent can protect themselves from these attacks with MEV-geth	Have you got any other feedback? (optional)
1	Strongly agree	Strongly agree	Strongly agree	Strongly agree	Great work, I think this could be improved by adding a few animations but apart from that it's great!
2	Strongly disagree	Strongly disagree	Strongly disagree	Strongly disagree	The colour scheme applied within the dashboard could be different. Specifically, it makes the data hard to grasp.
3	Strongly agree	Strongly agree	Strongly agree	Neutral	Great Work!!
4	Strongly agree	Neutral	Agree	Agree	Didn't really understand the Sandwich attacks, could have gone through an example.

Chapter 5

Conclusion

5.1 Contributions & Achievements

5.1.1 Asynchronous Web3 Software

This project contributes a fully fledged ecosystem for interacting with the blockchain asynchronously in Python. It extends the base Web3.py package with a set of object-orientated wrapper classes to abstract away much of the complexity when dealing with typical blockchain paradigms such as transactions, accounts and smart contracts. This software abstracts away the complexities of managing multiple in-flight concurrent pending TXs, implements rebroadcasting automatically by design and TX cancellations.

5.1.2 Front-running Attack Implementations & Mitigations

This project contributes implementations of generalised Displacement attacks, Sandwich attacks and PGAs in near real-life scenarios to exemplify the ease of implementation and high risk to users of blockchain systems. MEV-geth is also implemented in the underlying software to send TXs through a Flashbots Auction and is shown to protect agents from these mempool based attacks from other non-miner attackers.

5.1.3 Educational Dashboard

This project contributes a web dashboard found at `http://3.66.150.226` to provide education and information to students and other interested parties. The dashboard provides the ability for users to run experiments of Displacement and Sandwich attacks on the real Goerli test-net and PGAs on a local Ganache blockchain instance. The dashboard visualises the interactions between agents and attackers during these experiments and the financial outcome of the attacks on all parties. This dashboard also provides the ability for agents to use MEV-geth to send TXs and exemplifies the protections provided by MEV-geth from other non-miner attackers. This dashboard also provides information related to the types of experiments available and how they are set up. The underlying smart contracts are also displayed.
This dashboard provides value to the following parties:

1. Students & researchers looking to understand blockchain, the mempool and mempool based front-running attack real-life implementations in more depth.
2. Blockchain users trying to protect themselves from front-running attacks in everyday life and the mitigations available to them.
3. Smart contract creators looking into the vulnerabilities in their smart contracts they need to avoid to prevent these attacks from occurring.

On the 6th of May 2022, a presentation to the Financial Technology with Data Science MSc cohort during the Financial Technology Group Project lab session was given to introduce these attacks and the dashboard. The feedback was positive and the dashboard may be used as a teaching resource in the future. A recording of this presentation can be seen at `https://youtu.be/AQ_8G7B1Q7Q`. A review and validation of the dashboard's utility as an educative tool are explored in Section 4.5.

5.2 Value & Impact

This project exemplifies the ease at which these attacks can be implemented in real-world scenarios by agents of the system with limited resources. After reading this paper, blockchain agents who want to execute an atomically profitable TX should think twice before sending the TX through the traditional mempool based approach; these agents should definitely use MEV-geth, or another applicable mitigation, to protect themselves from these attacks.

This project should encourage smart contract developers to think twice about front-running vulnerabilities when creating their applications. It also bridges the gap between the theoretical depiction of these attacks and actual implementation, allowing the developers to test the front-running protections they have created.

This project outlines the significant security issues still facing blockchain concerning MEV & front-running attacks despite the introduction of MEV-geth. It should encourage Flashbots and other researchers to continue iterating upon the MEV-geth protocol to remove the risk still present of malicious miners, as outlined in Section 4.3.8.

The dashboard created during this project provides a model example of these attacks to be used for insight into practical implementations of front-running in blockchain environments. This dashboard can be used in academic and educational settings. The dashboard may be used by the University of Bristol as a teaching resource in the future.

5.3 Project Status

The underlying Web3 interactivity software is a fully fledged ecosystem of object-orientated wrappers on top of Web3.py in Python; it is working as-is and can be utilised as the underpinnings of any Web3 interactivity software in Python.

As covered in Section 4.4.1, basic generalised Displacement, Sandwich and PGA attackers are fully implemented in an experimental environment with the caveat of using the internal TX mempool for mempool analysis due to Infura node constraints. If the mempool issue were solved either through the use of a different node provider or through running a personal node, this basic system would work as-is in a production environment.

As covered in Section 4.4.2, MEV-geth is implemented into the system as an option for sending TXs. Agents can use the system to utilise MEV-geth on Ethereum blockchains supporting it to mitigate themselves from non-miner front-running attacks as-is.

As covered in Section 4.4.3, the dashboard available at `http://3.66.150.226` is a fully fledged environment for learning about and running experimental Displacement, Sandwich and PGA attacks as an educative tool, both through traditional and MEV-geth TXs.

5.4 Future Work

To align the attack implementations as close to real conditions as possible, there are two required further extensions:

1. As mentioned in Section 4.3.1, rather than utilising the internal TX pool as a representation of the mempool, visibility into the real mempool must be established on real networks using a different node provider or a dedicated local node.

2. The PGA experiment should be implemented on the Goerli Network using the limit on gas fees as outlined in Section 4.3.5 to prevent excessive ETH usage. Running the experiment on the Ganache local blockchain removes some of the potential intricacies of the experiment being run in a decentralised system.

Whilst not absent like the steps previously mentioned, the following would be valuable additions and extensions of the work:

1. More experiment types could be implemented for further insight and education value. Attacks such as Suppression Attacks (Section 2.2.4) and Sybil Attacks (Section 2.2.5) could be implemented as these are also common in-flight TX attacks. These attacks were not implemented due to time constraints.

2. Whilst not a requirement for a basic implementation of a generalised front-runner, as mentioned in Section 4.3.2, contract decoding without previous knowledge of the contract (i.e. without access to ABI) would allow a more complex Displacement implementation to be developed. If all contracts could be decoded, Displacement attackers could simulate different combinations of replaced address information parsed to contract methods as parameters. At a minimum, the workaround to obtain verified smart contract ABIs using the Etherscan API endpoint [23] would enable this on at least a portion of unknown contract interactions in the mempool.

3. Upgrading the method used for simulating TXs would be a desirable improvement. As mentioned in Section 4.1, both forking and simulating TXs in a sandbox environment take up significant time. The current methodology would only be able to compute a fraction of TXs per block in a production system. Improvements could include: maintaining the current ganache fork, rebasing it on new blocks for each block simulation round; batching simulated TXs together when simulating multiple in a block, some method would be needed to separate the outcomes of different TXs or potentially using a local node instead for simulating TXs, discarding Ganache entirely.

4. As mentioned in Section 4.5, a real-time mempool visualisation on the dashboard would provide additional insight into the workings of the experiments to the user. If this were implemented, users would be able to see the attacker's TXs enter the pool after the agent's, and see they have higher or lower priority based off of their gas prices.

5. This project has specifically focused on in-flight atomically profitable TXs. Current blockchain state analysis could itself lead to profitable opportunities attackers could then seize using MEV-geth. As mentioned in Section 4.3.6, this could even apply to the holder contract created in this project: an opportunity arises the instant a `receiveFunds()` TX is completed.

5.5 Conclusion

This project has analysed in-flight front-running attacks through an implementation-specific lens. This project presents implementations for common front-running attacks and how non-miner attackers can be mitigated through the use of MEV-geth. It also provides easy to use asynchronous software to interact with Ethereum blockchains in Python to extend this work further or for anything else requiring blockchain interaction. The dashboard this project has produced has been validated as a useful educative tool that may be used by the University of Bristol as a teaching tool in the future.

Bibliography

[1] 1inch. 1inch. https://app.1inch.io/. Accessed: 21/04/2022.

[2] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. https://uniswap.org/whitepaper-v3.pdf, 2021.

[3] Alchemy. Goerli faucet. https://goerlifaucet.com/. Accessed: 02/05/2022.

[4] Carsten Baum, James Hsin yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. Sok: Mitigation of front-running in decentralized finance. https://eprint.iacr.org/2021/1628, 2021.

[5] James Beck and Mattison Asher. What is eip-1559? how will it change ethereum? https://consensys.net/blog/quorum/what-is-eip-1559-how-will-it-change-ethereum/. Accessed: 25/04/2022.

[6] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf, 2014.

[7] Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. Eip-1559. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md. Accessed: 25/04/2022.

[8] Celery. React. https://docs.celeryq.dev/en/stable/index.html. Accessed: 08/05/2022.

[9] BitInfo Charts. Ethereum avg. transaction fee historical chart. https://bitinfocharts.com/comparison/ethereum-transactionfees.html#3y. Accessed: 25/04/2022.

[10] Jiachi Chen, Xin Xia, David Lo John Grundy, and Xiaohu Yang. Maintaining smart contracts on ethereum: Issues, techniques, and future challenges. https://arxiv.org/abs/2007.00286, 2021.

[11] Coinbase. Around the block #3: analysis on the bzx attack. https://blog.coinbase.com/around-the-block-analysis-on-the-bzx-attack-defi-vulnerabilities-the-state-of-debit-cards-in-1289f7f77137. Accessed: 08/05/2022.

[12] CoinMarketCap. All cryptocurrencies. https://coinmarketcap.com/all/views/all/. Accessed: 05/04/2022.

[13] Colblindor. Deuteranopia - red-green color blindness. https://www.color-blindness.com/deuteranopia-red-green-color-blindness/. Accessed: 07/05/2022.

[14] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. https://arxiv.org/abs/1904.05234, 2019.

[15] Chris Dannen. Introducing ethereum and solidity. https://link.springer.com/content/pdf/bfm%253A978-1-4842-2535-6%252F1.pdf, 2017.

[16] Haitz Sáez de Ocáriz Borde. An overview of trees in blockchain technology: Merkle trees and merkle patricia tries. https://www.researchgate.net/profile/Haitz_Saez_De_Ocariz_Borde/publication/358740207_An_Overview_of_Trees_in_Blockchain_Technology_Merkle_Trees_and_Merkle_Patricia_Tries/links/6212f4aff02286737cb1168d/An-Overview-of-Trees-in-Blockchain-Technology-Merkle-Trees-and-Merkle-Patricia-Tries.pdf, 2022.

[17] Ethereum. Account types, gas, and transactions. https://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html. Accessed: 15/04/2022.

[18] Ethereum. Erc-20 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/. Accessed: 15/04/2022.

[19] Ethereum. Go ethereum. https://github.com/ethereum/go-ethereum. Accessed: 21/04/2022.

[20] Ethereum. The merge. https://ethereum.org/en/upgrades/merge/. Accessed: 23/04/2022.

[21] Ethereum. Solidity documentation. https://docs.soliditylang.org/en/latest/. Accessed: 07/04/2022.

[22] EtherScan. Ethereum gas tracker. https://etherscan.io/gastracker. Accessed: 23/04/2022.

[23] EtherScan. Get contract abi for verified contract source codes. https://docs.etherscan.io/api-endpoints/contracts#get-contract-abi-for-verified-contract-source-codes. Accessed: 02/05/2022.

[24] eth.wiki. Rlp. https://eth.wiki/en/fundamentals/rlp. Accessed: 15/04/2022.

[25] Gianni Fenu, Lodovica Marchesi, Michele Marchesi, and Roberto Tonelli. The ico phenomenon and its relationships with ethereum smart contract environment. https://ieeexplore.ieee.org/abstract/document/8327568, 2018.

[26] Flashbots. Introduction. https://explore.flashbots.net/data-metrics. Accessed: 23/04/2022.

[27] Flashbots. Mev-geth. https://github.com/flashbots/mev-geth. Accessed: 21/04/2022.

[28] Flashbots. v0.6. https://docs-staging.flashbots.net/flashbots-auction/miners/mev-geth-spec/v06.

[29] Flashbots. Welcome to flashbots. https://docs.flashbots.net/. Accessed: 23/04/2022.

[30] Ethereum Foundation. eth-account. https://github.com/ethereum/eth-account. Accessed: 07/05/2022.

[31] Liam Frost. Was the $1 million defi hack illegal or not? https://decrypt.co/20512/was-the-1-million-defi-hack-illegal-or-not, 2020. Accessed: 06/04/2022.

[32] Gervais, Arthur, Karame, Ghassan O., Wüst, Karl, Glykantzis, Vasileios, Ritzdorf, Hubert, and Srdjan Capkun. On the security and performance of proof of work blockchains. https://dl.acm.org/doi/abs/10.1145/2976749.2978341, 2016.

[33] PostgreSQL Global Development Group. Postgresql. https://www.postgresql.org/. Accessed: 08/05/2022.

[34] Lioba Heimbach and Roger Wattenhofer. Eliminating sandwich attacks with the help of game theory. https://arxiv.org/abs/2202.03762, 2022.

[35] Lioba Heimbach and Roger Wattenhofer. Sok: Preventing transaction reordering manipulations in decentralized finance. https://arxiv.org/abs/2203.11520, 2022.

[36] Docker Inc. Docker. https://www.docker.com/. Accessed: 08/05/2022.

[37] Infura. Txpool methods not allowed. https://github.com/INFURA/infura/issues/55. Accessed: 08/05/2022.

[38] Don Johnson and Alfred Menezes. The elliptic curve digital signature algorithm (ecdsa). https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.8014, 1999.

[39] Sunny King and Scott Nadal. On the security and performance of proof of work blockchains. https://bitcoin.peryaudo.org/vendor/peercoin-paper.pdf, 2012.

[40] Logilab. unittest - unit testing framework. https://github.com/PyCQA/pylint. Accessed: 02/05/2022.

[41] Joshua Mapperson. Ethereum's top 10 dapps hit 1m users this month. https://cointelegraph.com/news/ethereum-s-top-10-dapps-hit-1m-users-this-month. Accessed: 07/04/2022.

[42] Meta. React. https://reactjs.org/. Accessed: 08/05/2022.

[43] Microsoft. Microsoft forms. https://forms.office.com/. Accessed: 08/05/2022.

[44] MistX. Mistx. https://mistx.io/. Accessed: 21/04/2022.

[45] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://www.ussc.gov/sites/default/files/pdf/training/annual-national-training-seminar/2018/Emerging_Tech_Bitcoin_Crypto.pdf, 2008.

[46] Svetlin Nakov. Practical cryptography for developers. https://cryptobook.nakov.com/, 2022. Accessed: 08/05/2022.

[47] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? https://arxiv.org/abs/2101.05511, 2021.

[48] Dan Robinson and Georgios Konstantopoulos. Ethereum is a dark forest. https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest, 2020. Accessed: 05/04/2022.

[49] Django Software Foundation. Django. https://www.djangoproject.com/. Accessed: 08/05/2022.

[50] Python Software Foundation. unittest - unit testing framework. https://docs.python.org/3/library/unittest.html. Accessed: 08/05/2022.

[51] Zachary Stucke. Goerli eth for uni project - anyone got any they could send? https://www.reddit.com/r/ethdev/comments/t8mdme/goerli_eth_for_uni_project_anyone_got_any_they/. Accessed: 02/05/2022.

[52] Nicolas T. Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in bitcoin digital currency. https://arxiv.org/abs/1402.1718, 2014.

[53] Alan T. Sherman, Farid Javani, Haibin Zhang, and Enis Golaszewski. On the origins and variations of blockchain technologies. https://ieeexplore.ieee.org/document/8674176, 2019.

[54] Polygon Technology. Ethereum-polygon bridge. https://docs.polygon.technology/docs/develop/ethereum-polygon/getting-started/. Accessed: 06/04/2022.

[55] Polygon Technology. Polygon sidechain. https://polygon.technology/. Accessed: 06/04/2022.

[56] Uniswap. Uniswap protocol. https://uniswap.org/. Accessed: 02/05/2022.

[57] Zurich University. Nft explosion: why are people buying digital art? https://www.futurity.org/nfts-nft-art-2671602-2/, 2021. Accessed: 07/04/2022.

[58] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges. https://arxiv.org/abs/2105.07447, 2021.

[59] Bitcoin Wiki. Secp256k1. https://en.bitcoin.it/wiki/Secp256k1, 1999. Accessed: 16/04/2022.

[60] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, 2022.

[61] Liyi Zhou, Kaihua Qin, Antoine Cully, Benjamin Livshits, and Arthur Gervais. On the just-in-time discovery of profit-generating transactions in defi protocols. https://arxiv.org/abs/2103.02228, 2021.

Appendix A

Presentation Report

Department of Computer Science

Theodoros Constantinides
Teaching Associate in Fintech

Merchant Venturers Building,
Woodland Road, Bristol, BS8 1UB
Email: tc17231@bristol.ac.uk

7th May 2022

To Whom It May Concern,

I had the pleasure to attend Zak Stucke's presentation on "Generalised Front-Running Attacks in Blockchain", which he game to the Financial Technology with Data Science MSc cohort during the Financial Technology Group Project lab session on the 6th of May 2022. During the presentation, Zak first showed why such attacks are possible, then introduced displacement and sandwich attacks, and priority gas auctions and finally discussed why the Flashbots Ethereum node implementation provides a potential mitigation to front-running attacks. At the end of the presentation, Zak demonstrated both attacks through an interactive dashboard that he built.

The session was really well planned and delivered. Zak demonstrated an in-depth understanding of the topic, and in a short time managed to introduce the audience to the key ideas and problems of MEV and front running attacks. Moreover, Zak's dashboard provides an excellent resource for teaching such topics, as it provides a way to visually show the attacks taking place in real time using a testnet and provides short descriptions explaining how the attacks operate. Although the topics of Zak's presentation are not covered directly in our unit, they are important for our students and his dashboard could be useful for our teaching in the future.

Yours faithfully,

Theodoros Constantinides
Teaching Associate in Financial Technology
Department of Computer Science, SCEEM
University of Bristol

Figure A.1: Feedback from the Teaching Associate Theodoros Constantinides, who was present during my presentation, to the Financial Technology with Data Science MSc cohort.

Appendix B

Smart Contracts

B.1 Holder Smart Contract

```
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract Displacement {
    using SafeMath for uint256;

    // immutable so they cannot be changed after constructor:
    address public immutable AGENT;
    address public immutable ATTACKER;

    constructor(address agent, address attacker) {
        AGENT = agent;
        ATTACKER = attacker;
    }

    modifier onlyAllowed {
        // Allow either the agent or attacker to withdraw:
        require(msg.sender == AGENT || msg.sender == ATTACKER, "Only agent and attacker
            can interact!");
        _;
    }

    // Allow receipt of funds:
    function receiveFunds() payable public {}

    function withdraw() public onlyAllowed {
        uint curBal = address(this).balance;
        require(curBal > 0, "Nothing to withdraw!");

        payable(msg.sender).transfer(curBal);
    }
}
```

Listing B.1: The holder contract used during Displacement and PGA experiments.

B.2 Pool Smart Contract

```
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract Sandwich {
    using SafeMath for uint;

    address public immutable AGENT;
    address public immutable ATTACKER;

    uint public POOL_TOKEN_BALANCE = 5 * 10**17; // Funded with 0.5 eth so identical
        balances initially
    uint public AGENT_TOKEN_BALANCE = 0;
```

```
13      uint public ATTACKER_TOKEN_BALANCE = 0;
14      uint public K_VAL; // Calculated on eth funding to set curEth * POOL_TOKEN_BALANCE =
            K_VAL
15
16      modifier onlyAllowed {
17          require(msg.sender == AGENT || msg.sender == ATTACKER, "Only agent and attacker
                can interact!");
18          _;
19      }
20
21      constructor(address agent, address attacker) {
22          AGENT = agent;
23          ATTACKER = attacker;
24      }
25
26      function updateUserTokenBal(uint newAmount) internal {
27          if (msg.sender == AGENT) {
28              AGENT_TOKEN_BALANCE = newAmount;
29          } else {
30              ATTACKER_TOKEN_BALANCE = newAmount;
31          }
32      }
33
34      // Allow receipt of funds + sets up the pool equation with the newly received eth:
35      function receiveFunds() payable public {
36          K_VAL = address(this).balance * POOL_TOKEN_BALANCE;
37      }
38
39      // Send eth to contract to swap with the made up internal "token":
40      function swapEthForTokens() public payable onlyAllowed {
41          uint newEth = msg.value;
42          require(newEth > 0, "No ether sent!");
43
44          // (newEth + curEth) * (POOL_TOKEN_BALANCE - x) = K_VAL
45          uint curEth = address(this).balance;
46          uint tokensOut = POOL_TOKEN_BALANCE.sub(K_VAL.div(newEth.add(curEth)));
47          POOL_TOKEN_BALANCE = POOL_TOKEN_BALANCE.sub(tokensOut);
48          uint oldUserTokenBal = msg.sender == AGENT ? AGENT_TOKEN_BALANCE :
                ATTACKER_TOKEN_BALANCE;
49          updateUserTokenBal(oldUserTokenBal.add(tokensOut));
50      }
51
52      // Swap all owned made up internal "tokens" back for eth:
53      function swapTokensForEth() public onlyAllowed {
54          uint newTokens = msg.sender == AGENT ? AGENT_TOKEN_BALANCE :
                ATTACKER_TOKEN_BALANCE;
55          require(newTokens >= 0, "No tokens for user to swap!");
56
57          // (curEth - x) * (POOL_TOKEN_BALANCE + newTokens) = K_VAL
58          uint curEth = address(this).balance;
59          uint ethOut = curEth.sub(K_VAL.div(POOL_TOKEN_BALANCE.add(newTokens)));
60          POOL_TOKEN_BALANCE = POOL_TOKEN_BALANCE.add(newTokens);
61          updateUserTokenBal(0);
62          payable(msg.sender).transfer(ethOut);
63      }
64
65      // Backup allow withdraw:
66      function withdraw() public onlyAllowed {
67          uint curEth = address(this).balance;
68          require(curEth > 0, "Nothing to withdraw!");
69
70          payable(msg.sender).transfer(curEth);
71      }
72  }
```

Listing B.2: The pool contract used during the Sandwich atacck experiment.

Appendix C

Transaction Monitor

```python
import asyncio
from datetime import timedelta
from web3.exceptions import TransactionNotFound

from django.utils import timezone

from dissertation.backend.asyncio_utils import indefinite_worker_wrapper
from dissertation.backend.tx.tx import TX_SUCCESS, TX_PENDING, TX_CANCELLED, TX_REVERTED
import dissertation.backend.tx.pool as tx_pool_holder


@indefinite_worker_wrapper
async def transaction_monitor():
    pending_txs = [tx for tx in tx_pool_holder.TX_POOL if tx.status == TX_PENDING]
    # Order the transactions by nonce:
    pending_txs = sorted(pending_txs, key=lambda tx: tx.data["nonce"])

    # Need to also separate by account:
    account_pks = set()
    for tx in pending_txs:
        account_pks.add(tx.account.pk)

    pending_txs_by_account = []
    for pk in list(account_pks):
        account_txs = [tx for tx in pending_txs if tx.account.pk == pk]
        pending_txs_by_account.append(account_txs)

    for account_pending_txs in pending_txs_by_account:
        for tx in account_pending_txs:
            # Check if any of the attempts have succeeded:
            at_least_one_hash_found = False
            tx_receipt_found = False
            # Putting tx_attempts inside list to prevent looping through new txs added to the pool (through e.g. a rebroadcast) in the same loop
            for index, tx_attempt in enumerate(list(tx.tx_attempts)):
                print(
                    "Checking tx for account: {}, nonce: {}, attempt: {}...".format(
                        tx.account.address, tx.data["nonce"], index
                    )
                )
                try:
                    info = await tx.account.net.provider.eth.get_transaction(
                        tx_attempt["hash"].hex()
                    )
```

```
                    print("Found tx hash!")
                    at_least_one_hash_found = True
                except TransactionNotFound:
                    info = None

                if info:
                    try:
                        # If this succeeds the transaction has completed
                            successfully!
                        tx_receipt = await tx.account.net.provider.eth.
                            get_transaction_receipt(
                            tx_attempt["hash"].hex()
                        )
                        tx_receipt_found = True

                        # Set the final value to the tx object:
                        tx.data = tx_attempt["data"]
                        tx.hash = tx_attempt["hash"]
                        tx.receipt = tx_receipt

                        # Check to see if the tx was reverted:
                        if tx_receipt["status"] == 1:
                            print("tx nonce: {} succeeded!".format(tx.data["
                                nonce"]))

                            if tx_attempt["is_cancel"]:
                                tx.status = TX_CANCELLED
                            else:
                                tx.status = TX_SUCCESS

                        else:
                            print("tx nonce: {} reverted!".format(tx.data["nonce
                                "]))
                            tx.status = TX_REVERTED

                        # Break from looping through the tx_attempts as one has
                            succeeded
                        break

                    except TransactionNotFound:
                        print("tx nonce: {} receipt not yet found.".format(tx.
                            data["nonce"]))

            print(at_least_one_hash_found, tx_receipt_found)
            if not at_least_one_hash_found:
                # Because of rebroadcast functionality, shouldn't disappear.
                # If the most recent tx is not in the pool within 10 seconds, tx
                     must have been dropped:
                if tx.tx_attempts[-1]["sent"] < timezone.now() - timedelta(
                    seconds=10):
                    # Resend the tx:
                    await tx.rebroadcast_if_needed(dropped=True)
            elif not tx_receipt_found:
                # Normal check to see if the tx needs rebroadcasting with a new
                    gas price if now too low:
                await tx.rebroadcast_if_needed()

    await asyncio.sleep(0.5)
```

Listing C.1: The background worker monitoring and rebroadcasting pending transactions.