IMPERIAL

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Blindly Backrunning Private Transactions With Fully Homomorphic Encryption

Author:
Majed Althonayan

Supervisor:
Dr Jonathan
Passerat-Palmbach

Submitted in partial fulfillment of the requirements for the MSc degree in Msc Computing (Security & Reliability) of Imperial College London

September 2024

Abstract

Blockchains and cryptocurrencies have experienced a monumental rise over the past decade. With Ethereum alone having around 1 million transactions per day [1], making it increasingly more attractive to opportunists who attempt to extract monetary value from transactions. This is, however, often at the expense of the user. As a result, it is of paramount importance to ensure that users are protected from malicious agents who exploit the public, transparent nature of Blockchains for individual gain.

A Blockchain is a chain of blocks, each of which consisting of transactions that are executed sequentially. The ability to alter this order of transactions (by insertion, removal and re-ordering of transactions) can lead to the extraction of additional value commonly referred to as Maximal Extractable Value (MEV). MEV has led to the extraction of $750 million from Ethereum before the merge [2]. Although certain forms of MEV are universally considered to have adverse effects on users and their experience, other forms of MEV, such as arbitrage and liquidations, are believed to have a positive effect in regulating the markets.

This research introduces a promising solution that allows searchers to backrun transactions, leveraging the effects of arbitrage while mitigating the harmful effects of MEV. It expands on the work done by Flashbots by utilising fully homomorphic encryption to enable the blind backrunning of transactions by searchers through the fhEVM framework [3] on the UniswapV2 decentralised exchange. This paper also addresses the challenges faced by previous works, aiming to reduce the computational overhead and enhance the solution's usability.

Despite computational constraints, this paper presents a novel solution to the outlined aims through the advancement of known solutions by allowing searchers to combine multiple transactions and accept a greater number of UniswapV2 methods, thereby allowing searchers to generate complex and novel arbitrage opportunities. This advancement is aided with use of the fhEVM framework [3] which was utilised to build and deploy the solution on the public network.

This paper represents a solid foundation for future research with the aim of further enhancing the use of fully homomorphic encryption in decentralised finance to create a fairer, more ethical ecosystem.

Acknowledgments

First and foremost, all praise be to God, the most compassionate, the most merciful. I thank God for granting me the health, guidance and ability to continue in my academic journey, and for allowing me to pursue my dreams surrounded by the greatest of people.

I would also like to express sincere gratitude to my supervisor, Dr Jonathan Passerat-Palmbach for his unwavering support, encouragement and advice. I would like to thank Dr Passerat-Palmbach for the great time and effort that he has invested into our meetings and his feedback, which provided me with great support and motivated me throughout this project. I would further like to extend my gratitude to Dr William Knottenbelt who, as my second supervisor, provided me with invaluable feedback, suggestions and ideas which were instrumental in shaping the quality of my research.

Moreover, I would like to thank my family for their unconditional love, support and patience that they have shown me during this endeavour. I would especially like to show the deepest gratitude to my parents. Their indispensable advice, unwavering support and unconditional love has been a cornerstone of my academic journey and motivated me to be the greatest version of myself. I would not have been able to do it without them.

Thank you all for being an integral part of this journey.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ABI Application Binary Interface

EVM Ethereum Virtual Machine

FHE Fully Homomorphic Encryption

fhEVM Ethereum Virtual Machine Using Fully Homomorphic Encryption

HE Homomorphic Encryption

MPC Multi Party Computation

OFA Order Flow Auction

PB Programmable Bootstrapping

PBS Proposer Builder Separation

PET Privacy Enhancing Technology

PHE Partially Homomorphic Encryption

P2P Peer-to-Peer

RLP Recursive Length Prefix

SHE Somewhat Homomorphic Encryption

SMPC Secure Multi Party Computation

TEE Trusted Execution Environment

TFHE Fully Fast Homomorphic Encryption Over The Torus

ThPKE Threshold Public Key Encryption

ZK Zero Knowledge

ZK-SNARK Zero Knowledge Succinct Non-Interactive Argument Of Knowledge

ZK-STARK Zero Knowledge Scalable Transparent Argument Of Knowledge

Chapter 1

Introduction

1.1 Blockchain Technology, Decentralised Finance, and Maximal Extractable Value

Blockchain technology, particularly the introduction of decentralised finance has revolutionised the financial and technological landscapes over recent years. By eliminating the need for traditional intermediaries, decentralised financing has allowed users to engage in financial transactions and access traditional financial services with the added benefit of decentralisation, transparency and openness. Blockchains collate user transactions in sequentially ordered blocks. In order to be included in these blocks, transactions must go through the operational supply chain, wherein parties have the ability to manipulate the order of transactions in a block. This capability has led to the emergence of maximal extractable value (MEV), a monetary value extracted from the block in addition to the normative block reward and transaction fees. Competition for MEV opportunities has led to higher gas fees, increased transaction delays, and a generally inferior user experience. Before detailing the intricacies of MEV, it is important to analyse the operational supply chain to a greater extent.

The Ethereum operational supply chain model, resulting from the proposer builder separation, consists of a number of actors including: users, searchers, builders and proposers. Users are the ultimate source of this supply chain where they create transactions by buying, selling or exchanging tokens [7]. Once validated, these transactions are published to every node on the Ethereum Network via the peer-to-peer gossiping protocol whereby valid transactions are added to a list of pending transactions commonly referred to as the mempool. Searchers scan these mempools to find user transactions that can be leveraged with other precisely crafted, searcher-generated transactions to extract additional value. Following this, searchers bundle multiple transactions by selecting the ones that alter the markets in the most profitable ways. Searchers then submit these bundles to block builders who utilise non-overlapping bundles as well as other individual transactions to create blocks that offer the most value. Builders then submit their own blocks to block proposers via relays, a trusted interface that verifies the validity of blocks ensuring that proposers

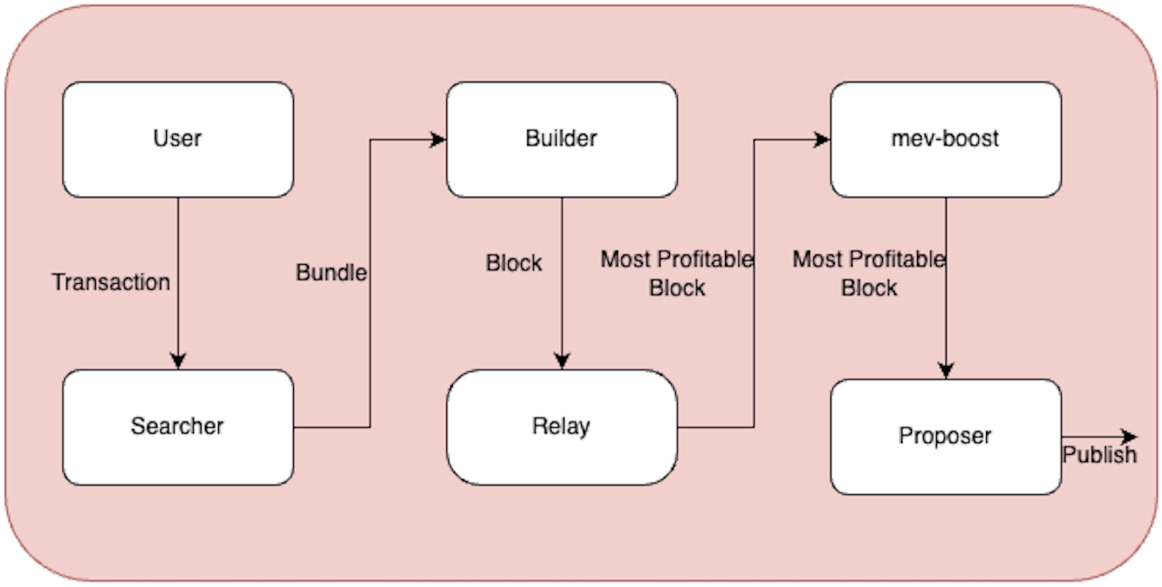

Figure 1.1: Transaction Supply Chain

do not get slashed[1] for signing invalid blocks [7]. The proposer for a given slot then selects the most profitable block from the ones submitted to them by the builders and submits it on the network. To automate this operation, proposers run the MEV-Boost software to process all received blocks and automatically submit the most profitable one [7]. In this life cycle of transactions, the extraction of this additional value and the construction of blocks have been entrusted to searchers and builders respectively. This outsourcing of roles to third parties is known as the proposer-builder separation (PBS) which is motivated from the recent concern of increased centralisation pressure on validators, resulting in only large entities being able to create blocks. As a result, this separation is utilised to lower the overhead required to be a validator, ensuring that even smaller entities are able to earn the same as large ones and therefore maintaining the decentralisation of Ethereum [7]. Searchers, builders and proposers are incentivised to build and submit the most profitable block as a result of the block reward, a financial remuneration for proposing valid blocks. As previously mentioned, searchers are able to extract further value from blocks by manipulating the order of transactions within a block. This value, which is in addition to the block reward, is known as MEV [7]. A high level overview of the outlined supply chain can be seen in Figure 1.1.

While MEV can be extracted at any step of the transaction supply chain, this study specifically focuses on the interaction between users and searchers, whereby searchers apply programs that act upon pending transactions of users to extract MEV. MEV consists of several forms including frontrunning (prioritising searcher transactions in a block), sandwich attacks (positioning searcher transactions around user transactions), Liquidations (paying off under-collateralized loans in exchange for a fee) and backrunning (arbitraging exchanges based on user transactions) [7]. Each

[1]Destruction / Confiscation of a validator's deposit as a penalty for acting dishonestly or maliciously.

form of MEV is specifically designed to exploit the market shifts caused by the user transaction in the most profitable way. Frontrunning and sandwich attacks are commonly thought of as being harmful to users for whom they create an unethical, unjust disadvantage. Arbitrage, on the other hand, is viewed more positively as a form of stabilisation for networks [8]. Arbitrage is also the most common variation of MEV, accounting for 47% of the profits, 68% of the transactions, and 82% of the bots [9]. This is why despite actively harming users and worsening user experience, this study will not focus on frontrunning or sandwich attacks. Rather, this study will focus on backrunning to allow searchers to continue in their use of arbitrage to service the network.

Given the uncensored and public nature of blockchains, there is currently no agreed upon solution for preventing frontrunning and sandwich attacks whilst also allowing arbitrage as a form of stabilisation. Early solutions attempted to protect against all forms of MEV (both useful and harmful) by encrypting the entire transaction. However, due to the importance of the core blockchain tenets, these solutions were not implemented, spurring the development of more recent work by Flashbots. Flashbots initially utilised Intel's SGX to allow searchers to blindly backrun private user transactions. However, due to the presence of covert channels being a threat to the confidentiality and privacy of users' transactions, Flashbots moved on to developing a searcher language using secure multi party computation (SMPC). SMPC is a tool that enables parties to compute on private inputs without revealing anything except the result [10]. Flashbots abstracted and simplified the problem by considering only one user, one searcher and one trusted builder. This study aims to address some of the shortcomings and open questions suggested in relation to the current status quo by building on the expressiveness and usability of the searcher language, as well as improving practicality in terms of reducing computational overhead. The details of the Flashbots solution as well as other solutions are analysed to a greater extent in Chapter 2.

1.2 Motivation

In addition to the increased transaction fees, security concerns and transaction delays, MEV also creates an unfair environment within Ethereum. This is since users do not receive any compensation or reward for the additional value extracted from their transaction. In the current state of MEV, the majority of the value extracted by searchers is accrued by validators.

A current solution which attempts to recompense users are order flow auctions (OFA) which allow searchers to bid on pending transactions for their exclusive use in extracting value. While this solution levels the playing field by rewarding users with a proportion of the value they helped create [11], it does not address the desire of users and searchers to keep their intellectual properties private. This imperative need for privacy has naturally led to the use of encryption and privacy enhancing technologies to provide a solution that allows searchers to extract MEV from user transactions without negatively affecting the user.

While existing papers discussed in Chapter 2 have sought to address these issues

in differing ways, they are limited with regard to their use cases and their computational overhead, meaning they are often not applicable in real world settings. This deficiency in the previous literature to provide a complete solution motivates this study, which attempts to rectify the aforementioned issues caused by MEV through using encryption with fully homomorphism, to allow searchers to blindly backrun user transactions.

1.3 Aims and Objectives

Aims

This study attempts to comply with the core blockchain tenets of transparency and security by utilising Fully Homomorphic Encryption (FHE) whose unique characteristic allow for calculations to be performed on encrypted data. This study will specifically focus on the UniswapV2 decentralised exchange to magnify the interaction between users and searchers in order to create a computationally feasible program capable of providing confidentiality for both parties, whilst continuing to allow searchers to execute their strategies to backrun user transactions. This research uses a framework that enables confidential smart contracts on the Ethereum Virtual Machine using FHE (fhEVM) [12] to design the protocol and searcher language. This study will demonstrate the design, implementation, optimisation and evaluation of this solution which aims to build upon the open questions posed by previous works to further advance the application of FHE to blockchain transactions through improvements to the expressiveness, usability and practicality of the backrunning protocol.

Objectives

The objectives of this study are to:

- Design a backrunning program that allows searchers to blindly backrun user transactions.
- Implement the designed program using Solidity and the fhEVM framework.
- Allow for searchers to successfully combine and ultimately backrun multiple user transactions.
- Backrun UniswapV2 transactions calling both the “swapExactETHForTokens” and “SwapExactTokensForETH” functions.
- Optimise the final solution to improve both the computational overhead and the run time of the protocol.
- Optimise security measures by utilising the fhEVM framework to ensure that both the user’s transaction and the searcher’s strategy remain private throughout the protocol.

- Deploy the protocol on the public network.

- Evaluate the protocol to ensure that it successfully backruns pending user transactions and returns a profitable searcher transaction.

1.4 Methodology

The research follows a five-phase methodology, as depicted in Figure 1.2. Each phase is specifically designed to tackle a unique objective, and to develop a sound and robust solution that meets the previously stated objectives.

Phase 1 entails a comprehensive review of existing literature and requirement analysis that extensively reviews the literature available on blockchain technologies, decentralised finance, privacy enhancing technologies and current solutions to MEV. Upon reviewing the existing literature, the requirements and challenges of each study is analysed further and utilised to define the scope and objectives of this study.

The second phase of the methodology is the design of the solution which builds upon previous solutions to incorporate the fhEVM framework as well as to ensure the utmost efficiency and robustness. This phase of the methodology focuses on designing a solution that meets the previously defined objectives as well as making sure that the design achieves high standards of security and performance.

Phase 3 is the implementation of the design, incrementally developing and integrating the system components to achieve the outlined goals. This phase includes setting up the development environment, developing an unencrypted version of the backrunning protocol then integrating the fhEVM framework to ensure the privacy of both the user transaction and searcher's strategy.

Phase 4 comprises of the comprehensive testing and experimentation of the solution with the aim of exposing missing functionalities, vulnerabilities and bugs. As potentially the most important phase of development, sound and complete testing is crucial to ensure the accuracy and reliability of the backrunning protocol as well as to identify the robustness and scalability of the solution.

The final phase of the methodology evaluates the resultant solution under differing conditions and loads to analyse the performance and security of the solution as well as to ensure that the solution meets all of the required objectives and functionalities. Following this, a more comprehensive evaluation is conducted to compare the developed solution with the existing solutions that aim to achieve the same objectives. The results of the evaluation will often result in returning to phase 2 of the methodology to re-design the solution based on the evaluated challenges. This iterated approach ensures a robust, efficient and scalable final product that builds upon previous solutions and addresses all of the detailed objectives.

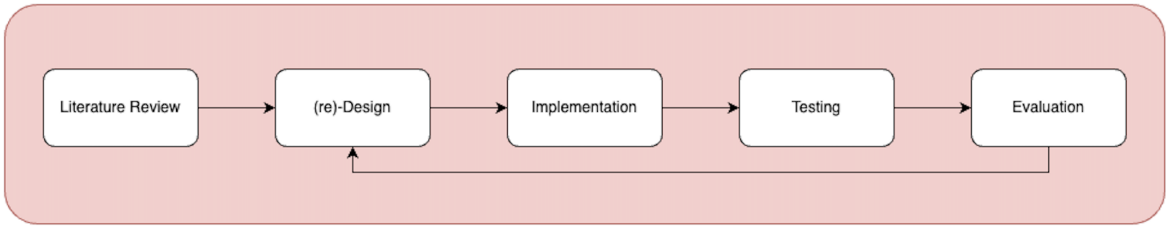

Figure 1.2: Abstracted Methodology

1.5 Contributions

This thesis aims to allow searcher to privately backrun multiple user transactions using Fully Homomorphic Encryption. In line with this aim, the following contributions are made:

1. **The Blind Backrunning of Private User Transactions Using FHE.**
   This paper provides a protocol that allows searcher's to backrun user transaction while ensuring that both their strategy as well as the user's transactions remain private throughout the execution.

2. **The Creation of Complex and Novel Strategies Through the Combination of Multiple User Transactions.**
   This paper acts as an extension of previous work by allowing searcher's to combine multiple transactions in an effort to find the most profitable combination.

3. **The Support of Multiple UniswapV2 Functions Including "swapExactEthForTokens" and "swapExactTokensForEth".**
   This paper provides support for user transactions trading both ETH for USDT as well as USDT for ETH. This acts as a further extension of previous solutions which were only able to support the former.

4. **The Deployment of the Backrunning Protocol on The Public Network.**
   This paper details the deployment on the backrunning protocol onto the public network. This is again an extension of previous work which deployed the backrunning protocol locally on the searcher's machine.

5. **The Experimentation and Analysis of the Backrunning Solution Under Differing Loads.**
   This paper showcases, the development, the experimentation and analysis of the protocol to ensure both efficiency and effectiveness.

6. **The Optimisation of the Backrunning Protocol To Achieve a Workable Solution.**
   This paper details the implementation of differing optimisation techniques to improve the protocol's gas consumption and overall runtime.

1.6 Structure of The Thesis

Chapter 1 of this paper introduces the context of the thesis by detailing the inner workings of Ethereum as well as describing the state of MEV in the broader context of decentralised finance. This chapter then moves on to outline the motivations, aims and objectives of this paper before outlining the abstracted methodology and research contributions.

Chapter 2 of this carried study sets the context for the rest of the paper by describing and evaluating the background theories and relevant literature. Chapter 2 begins by giving an overview of the current state of decentralised finance before evaluating the different privacy enhancing technologies including trusted execution environments, secure multi party computations, zero knowledge proofs and homomorphic encryptions. Finally, this chapter concludes by canvassing relevant solutions that aim to address similar objectives to those previously stated.

Following this, Chapter 3 moves on to outline the design, architecture and challenges of the proposed solution. It begins by outlining and justifying the design choices undertaken during development, including both the choices of encryption framework as well as the deployment setting. Following this, the chapter moves on to describe some of the anticipated challenges of the solution, including common issues faced within this context where efficient and effective solutions are often lacking. This chapter then introduces the design and architecture of the backrunning protocol by first detailing the high-level overview and workflow of the system before moving to define the intricate design and smart contract architecture of each method. This chapter concludes by introducing of the steps taken, both during and post development, to ensure that the proposed solution achieves a high-level of maintainability, ensuring both the robustness and scalability of the solution.

Chapter 4 begins to implement the designs outlined in Chapter 3. This Chapter first implements the transaction decoding contract detailing both the initial, manual decoding, as well as the final, automated, decoding. Following this, the chapter moves to describe the implementation of the contract that allows for the combination of several user transactions. Similarly, this section outlines both methods that were trialled during implementation. Chapter 4 then moves to outline the implementation of the main methods that perform the most essential part of the solution, namely, the trade size and profit calculations. Finally, this chapter concludes by describing the methods used to build and encode the final backrunning transaction utilising the previously calculated parameters.

Chapter 5 improves upon the initial implementation of the backrunning protocol by optimising the protocol to ensure an efficient and workable solution. Chapter 5 begins by introducing the need for optimisations. Following this, chapter 5 evaluates some of the most effective and prevalent optimisation techniques for reducing both the gas consumption and runtime of the backrunning protocol. The following section then moves to implement these techniques.

Chapter 6 introduces the tests and experiments performed on the final solution to measure the systems efficiency and effectiveness. This chapter begins by describing the experimental setup by first outlining the testing parameters. These include the test cases, the input parameters and the metrics to be evaluated. The next section

of this chapter then moves to outline and analyse the results of these experiments before concluding the chapter with an end-to-end walk-through of the backrunning solution.

Chapter 7 of this study evaluates the performance and implementation of the backrunning solution by first outlining an overview of the assumptions and limitations encountered during this project. Following this, it compares the developed backrunning protocol with previous solutions, highlighting both the similarities and differences.

The final chapter of this study concludes by first summarising the project's achievements. Chapter 8 then moves to address the legal, social, ethical and professional considerations related to the project, before concluding with a discussion of future developments and a general summary of this thesis.

Chapter 2

Background

The preceding chapter has moved to introduce the context of the thesis by detailing the inner workings of Ethereum as well as describing the state of MEV in the broader context of decentralised finance.

This Chapter, meanwhile, gives a more expansive and detailed account of the background theories, technical information and literature relevant to the objectives identified in Chapter 1. This chapter begins by introducing the state of decentralised finance on Ethereum before diving into privacy enhancing technologies (PETs) and finally a review of previous solutions.

Section 2.1 begins by introducing the concepts related to decentralised finance including the UniswapV2 decentralised exchange, MEV and Dark Pools. This is done to clearly outline the context in which the study will take place.

Section 2.2 introduces Key theories in privacy enhancing technologies that are used to implement the solutions discussed in section 2.3, which details the functionality, applications and challenges of trusted execution environments, secure multi-party computation, zero knowledge proofs and homomorphic encryption.

Section 2.3 canvasses the previous solutions that also aim to address similar objectives to those targeted by this research. This section outlines examples of dark pools as well as describing three solutions from Flashbots which utilise the Intel SGX, secure multi party computation and fully homomorphic encryption to implement their respective solutions.

2.1 Decentralised Finance

2.1.1 UniswapV2

On Ethereum, users are able to trade cryptocurrencies using either centralised or decentralised exchanges. Centralised exchanges operate similar to traditional stock exchanges where a central authority acts as an intermediary between buyer and seller. Decentralised exchanges, however, are peer-to-peer protocols where buyers interact directly with sellers without the need for a central authority controlling their funds. The UniswapV2 protocol is an example of a decentralised exchange available on the Ethereum blockchain that is designed for exchanging cryptocurrencies [13].

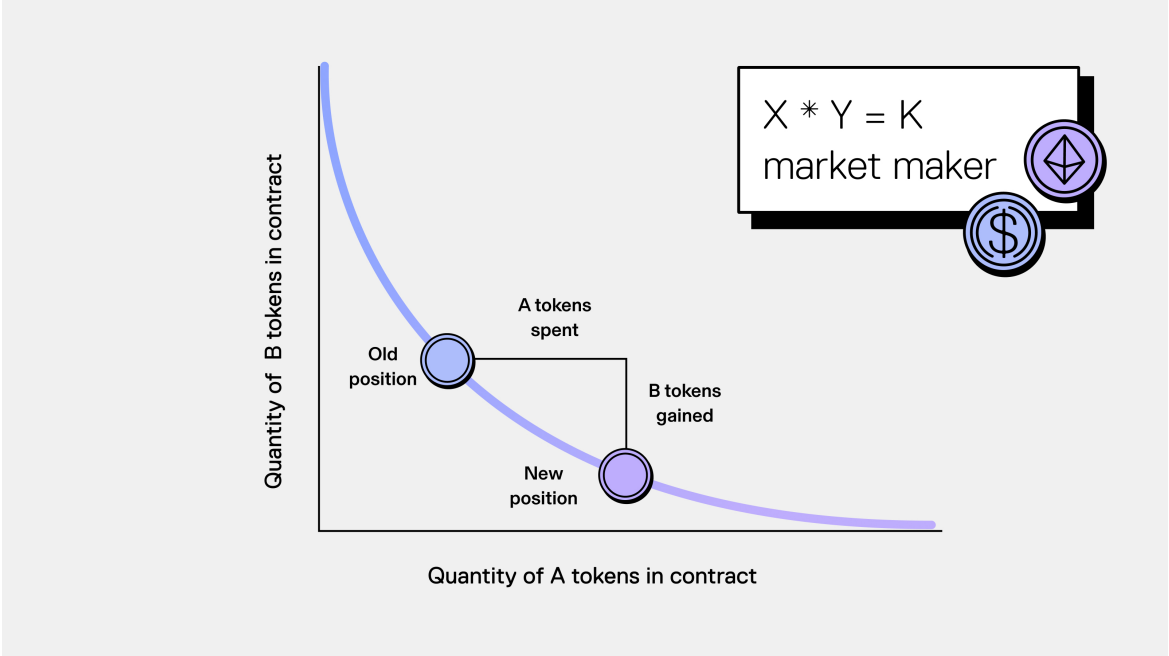

Figure 2.1: Uniswap Market Maker [4]

The Uniswap protocol is a set of smart contracts that are designed to "prioritize censorship resistance, security, self-custody, and to function without any trusted intermediaries who may selectively restrict access" [13].

Uniswap is an implementation of an automated liquidity protocol based on a constant product formula [14]. Each Uniswap pool stores reserves of two tokens where the quantities of these two tokens in the pool is described through the constant k which maintains their relationship. The price of each token is described by the pool's ruling equation $x \cdot y = k$ [15], where x and y are the respective quantities of token A and token B. This market maker mechanism, as seen in Figure 2.1, allows the reserves of tokens to remain balanced and in relative equilibrium with respect to each other. Given a user that wants to swap token A for token B on Uniswap, the quantity of token B received relative to the amount of token A spent is determined based on the current price of token B in the pool.

Once the swap is completed, the new values for the token reserves are now reflected on the chain, and the new token prices are immediately updated for all users as a result of the atomic nature of Ethereum transactions. It is important to note that all Uniswap transactions also incur a 0.3% transaction fee that is split by liquidity providers in proportion to the amount of liquidity that they have provided [16].

2.1.2 Maximal Extractable Value

As previously mentioned, the updated price of tokens in a pool is immediately updated for all users upon the completion of a trade. This price that is updated for all users is only true for that specific pair on Uniswap and is not necessarily true for other exchanges which will often have the same pair trading at different prices.

This discrepancy in price can be leveraged to produce risk free profit. This can occur when tokens are purchased at a cheaper exchange, before being sold at a more expensive exchange, with the difference in prices being the profit that the trader makes. This exploitation of price differences between markets is commonly known as arbitrage and is viewed positively as a result of its stabilising effects on the prices of tokens across discrepant exchanges.

The greater the number of traders that compete to seek arbitrage opportunities, the harder it becomes to find such opportunities due to the resolved price differences. This principle is known as the no-arbitrage principle, whereby tokens are priced in a way such that no trader is able to make risk-free profit. However, upon finding an arbitrage opportunity, traders must have knowledge of two things in order to effectively arbitrage this opportunity:

- The guarantee that the price of tokens will not change by the time they attempt to arbitrage the exchange.
- Knowledge of the quantity of tokens to trade (in order to ensure maximum profit).

As discussed in the previous chapter, transactions in a block are executed sequentially. Therefore, the only way to ensure that the prices of the tokens do not shift is to ensure that the trader's transaction is executed directly after the transaction that caused the initial arbitrage opportunity. This action of inserting a transaction after a target transaction is referred to as backrunning and acts as a means to arbitrage. As for the amount of tokens that the trader needs to swap to exploit the opportunity, this is calculated using the ruling equation of the relevant decentralised exchange. This calculation is discussed in greater detail in the following chapter.

Frontrunning is another form of MEV where the insertion of the trader's transaction is done before a user's transaction rather than after. In the event that the user's transaction is profitable, the trader's transaction will be executed first, thus capturing the profit before the user's transaction is able to execute. Similarly, another form of MEV is a sandwich attack which is when the trader simultaneously frontruns and backruns the user, surrounding the user's transaction so that they are able to capture the user's profitable transaction as well as arbitrage the price difference caused by the first two transactions. Both frontrunning and sandwich attacks have negative, unethical consequences on users and thus this study will not focus on either one but rather will focus on backrunning and arbitrage to ensure work undertaken based on this paper remains ethical and just. A comparison of frontrunning, backrunning and sandwich attacks within a block can be seen in Figure 2.2.

2.1.3 Dark Pools

As with arbitrage, dark pools similarly aid in stabilising markets without negatively affecting users. Dark pools are private exchanges used for trading tokens and they act as a solution to mitigate the price shifts caused by large bids, a common problem in traditional exchanges. Dark pools allow investors to place large orders without

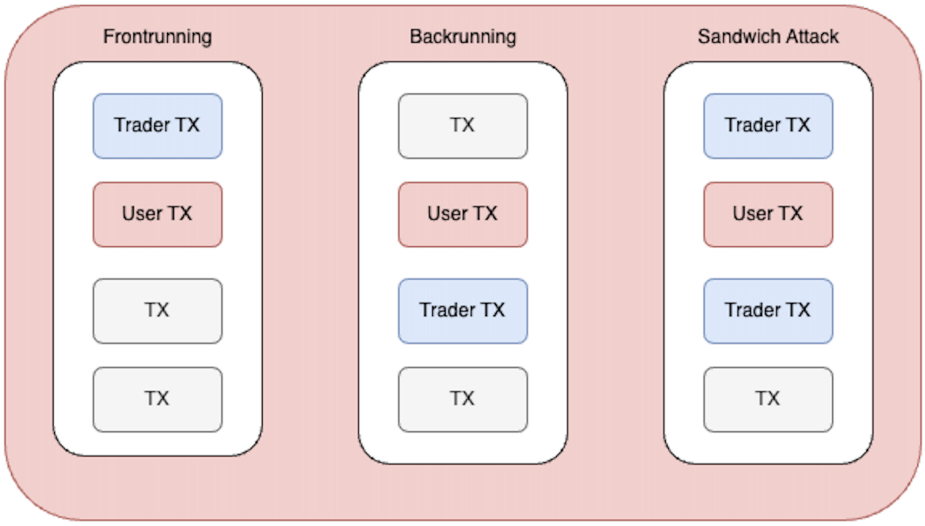

Figure 2.2: Maximal Extractable Value

notifying the market of their intention or of their trading strategy [17] thus minimising the price shift caused by the large bid. Traditionally, dark pools were viewed as an exclusive trading venue where only their operator's have full knowledge of the trading order book. This solution allowed participants to trade large orders without the fear of slippage or of other participants being aware of their bid or strategy. However, a shortfall of traditional dark pools is that they necessitated the trust that operator's would not act upon or publish their knowledge of the order books [17]. Within the context of the blockchain and in keeping with its trustless, decentralised nature, dark pools were built in such a way that even their operators had no knowledge of the orders until they had been successfully matched. This solution is referred to as "blind matching" [17].

Since dark pools mitigate the price shifts caused by large bids, they are able to effectively avoid frontrunning as well as allow maker orders to occur without slippage [18]. This limiting of slippage and price volatility will in turn cause arbitrage opportunities to diminish due to the stabilised nature of prices across exchanges.

Dark Pools vs Backrunning

Despite having similar objectives, dark pools differ from backruns in a plethora of ways. Dark pools aim to minimise price shifts by providing a private trading venue whereas backruns leverage price shifts by arbitraging pending transactions for profit. Additionally dark pools attempt to protect trading strategies by anonymising trades whereas backruns often exploit the trading strategies of users often without consent. Some examples of Dark Pools implemented by ZAMA, Sunscreen and Renegade are discussed in greater detail in section 2.3.

As a means to arbitrage, backrunning provides a benefit to all. Searchers gain significant profits from exploiting price imbalances, markets benefit as a result of the increased liquidity and price stabilisation while users are able transact with lower transaction fees and improved delays. As a result of these advantages, backrunning

is generally viewed positively. However, unlike backrunning, dark pools are more of a controversial technology. The existence of dark pools reflects the frustrations between market efficiency, transparency and fairness. Some view the technology positively as a beacon for innovation, allowing for improved transaction fees, reduced reporting and disclosure requirements and fewer intermediaries. Some would also make the argument that increasing the liquidity in dark pools introduces the capital necessary to "calm the waters of an industry plagued with fear and uncertainty" [18]. However, others do not view the introduction of dark pools positively. With increasing concerns over the lack of transparency potentially leading to reduced trust and market manipulation, which may further introduce regulatory scrutiny to police unfair trading practices. These controversies and concerns associated with dark pools have pushed this project towards the direction of backrunning as a means to arbitrage and price stabilisation. The universal benefit gained by backrunning in addition to the minimal concerns combine to make backrunning ideal to mitigate MEV.

2.2 Privacy Enhancing Technologies

2.2.1 Trusted Execution Environments

A key technology that is often used in this context are Trusted execution environments (TEE), which are tamper resistant processing environments that run on a separation kernel [19]. In addition to protecting the runtime state, TEEs also protect the stored assets as well as guaranteeing the authenticity of executed code, the integrity of runtime states and the confidentiality of code and data stored on persistent memory. TEEs use encryption to achieve this by segregating and protecting areas of memory and CPU from the rest of the CPU so that it cannot be read or tampered by anything outside of the segregated environment. This allows code executing inside the enclave to be processed in the clear while code outside of it needs to be suitably authorised [20]. Intel's SGX utilises a TEE to protect data while it is actively in memory via an "application isolation technology" [21]. Given that TEEs are isolated from the rest of the platform, the contents can be protected from observation and tampering through the use of hardware memory protection and cryptographic protections [19]. This feature of TEEs and Intel SGX allows it to be perfect for implementing a solution for MEV, an example of this can be seen in the following section.

TEEs' ability to provide high level security makes them amenable to utilisation in a wide range of settings. In addition to possible uses in mitigating MEV, TEEs can be used for ticketing, online transactions , cloud storage services among other applications [19]. One common shortfall of TEEs is that the security infrastructure upon which it depends tends to be susceptible to covert channel attacks, whereby attackers utilise extra information such as CPU usage or memory page allocation to reveal information about the data within the enclave.

2.2.2 Secure Multi-Party Computation

Secure multi-party computation (SMPC) is an evolution of Shamir's secret sharing scheme which distributes unique parts of the private key amongst users to ensure that no one is able to see the full secret by themselves but, only as a result of combining all the other shares. This, therefore, offers users a protocol that protects data during collaborative computations [22]. In SMPC, since participating parties are not involved in the computations and only the final result is public, it is ideal for tackling MEV as the data can remain private until the final backrunning transaction is deployed on chain. Furthermore, compared to FHE, SMPC has a lower computational overhead thus inducing less stress on the system [22]. This, combined with the increased take up in emerging technologies has allowed SMPC to be used as a tool for computations especially on private data where it has an advantage in solving security and privacy issues [23].

These advantages allow SMPCs to be used for a plethora of applications, such as key management and protection solutions where it preferred over other PETs such as homomorphic encryption as a result of its use of traditional AES, allowing it to be "more accessible and user-friendly while also keeping data secure" [22]. Despite its advantages, SMPCs suffers some limitations such as its vulnerability to willing collusion between parties, where if two parties are willing to share and combine their secrets, they can then be able to deduce the data of a third party. However, this limitation can be mitigated through the use of privacy zones which are created by utilising multiple domains each of which assigned to a separate user with their own privacy restrictions. This separation stops parties from colluding and thus protects the data.

2.2.3 Zero Knowledge Proofs

Zero Knowledge (ZK) Proofs are a way of proving knowledge about a piece of data without having to reveal the data itself. ZK Proofs were first introduced by Shafi Goldwasser, Silvio Micali and Charles Rackoff in 1985 who demonstrated that they could prove to another party that something was true without revealing any information apart from the fact that this specific statement is true [24].

In order to make ZK proofs possible, an algorithm must take some data as input and only return true or false making sure to not reveal the input data. This algorithm must satisfy the following criteria [24]:

1. **Completeness**: Given a valid input, the protocol always returns "True". Therefore, If the claim is true, and both parties are acting honestly then the proof can be accepted.

2. **Soundness**: Given an invalid input, it is impossible to make the protocol return 'True'. Therefore a lying claimant cannot trick an honest verifier into believing a false claim.

3. **Zero-Knowledge**: The verifier cannot learn anything about the statement except its validity. This is done to stop the verifier from deriving the claim from

the proof.

One variation of ZK proofs is the Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZK-SNARK) which in addition to the previous criteria, ZK-SNARKs are also succinct (have smaller ZK-proofs), non-interactive (the prover and verifier only interact once), and of Knowledge (ZK-proof cannot be constructed without access to the secret information) [24]. Another similar version is Zero-Knowledge Scalable Transparent Argument of Knowledge (ZK-STARK) which is faster at generating and verifying proofs and unlike ZK-SNARKS, relies on publicly verifiable randomness to generate public parameters for proofs and verifications [24].

Some limitations of ZK-proofs are their use of complex calculations, necessitating specialised machines which have significant costs, thus hindering their use with the majority of users. Furthermore, since protocols such as ZK-SNARKs utilise elliptic curves for encryption they are not quantum secure, making their attack surface larger than ZK-STARKs which, on the other hand, are considered quantum secure since they rely on collision resistant hash functions [24].

2.2.4 Homomorphic Encryption

Homomorphic Encryption is a form of encryption that permits computations to be performed on ciphertexts without the need for decryption. A compelling feature of homomorphic encryption is its ability to maintain the integrity of operations. As can be seen in Figure 2.3, homomorphic encryption can guarantee that the result of a computation on encrypted data is equivalent to the encryption of the same computation but performed on unencrypted data.

There are three main variations to homomorphic encryption: partially homomorphic encryption (PHE), somewhat homomorphic encryption (SHE) and fully homomorphic encryption (FHE). While PHE allows for an unlimited number of operations it only allows for one mathematical function to be performed. On the other hand, while SHE does support multiple mathematical functions, it is limited in the number of operations that can be performed. Given the limitations of both PHE and SHE, this study focuses on FHE which is able to support multiple operations and is not limited in the number of operations that can be performed.

Before proceeding to outline work utilising Fully Homomorphic Encryption, it is important to detail the technical characteristics of Fully Homomorphic Encryption. FHE enables the processing of encrypted data, allowing computations including addition and multiplication to be performed on them [25]. The first construction of this was in 2009 by Craig Gentry, who demonstrated an arbitrary amount of addition and multiplication [25]. FHE works by adding noise to encrypted data to ensure appropriate levels of security. The problem is that as the number of computations begins to grow so does the added levels of noise, meaning that regardless of the amount of space left for noise to grow into, the number of computations will always be limited by design. This motivated the use and development of bootstrapping which is a "special operation that resets the noise to its nominal level" [25] thus removing the limits imposed by the noise with the cost of additional memory and computational overhead.

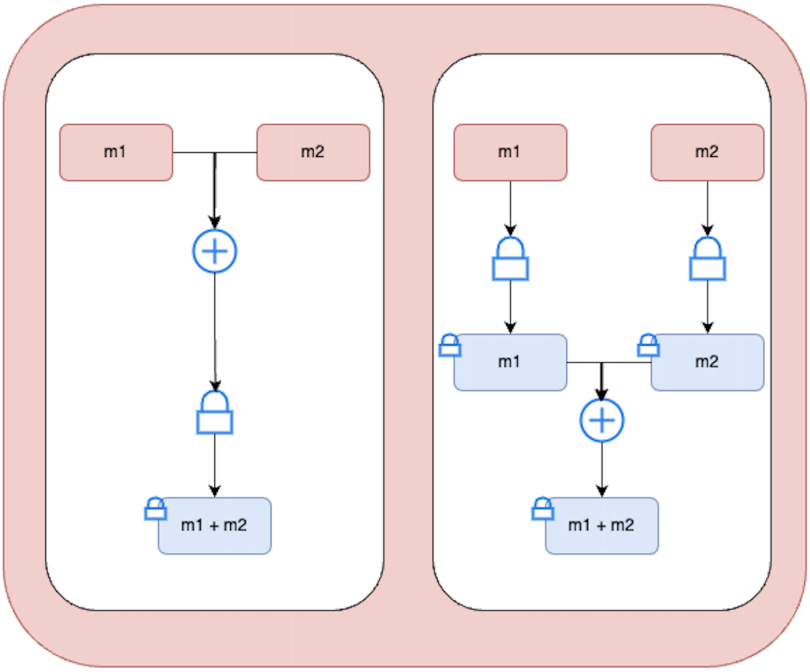

Figure 2.3: Homomorphic Encryption

Alternatively, another popular FHE scheme, utilised by many frameworks including the fhEVM framework is the Fully Fast Homomorphic Encryption over the Torus (TFHE) scheme. TFHE addresses the computational overhead caused by bootstrapping and allows for an infinite number of computations, exact arbitrary amounts and fast bootstrapping. TFHE utilises a homomorphic lookup table to remove any limits or approximations by encoding functions as a lookup table [25].

Threshold Fully Homomorphic Encryption

In public key encryption, only one party needs to have knowledge of the secret key in order to decrypt a message that has been encrypted by the corresponding public key. Due to possible corruption or loss of the key, having a single point of failure is a liability to the secureness of the scheme. This vulnerability has motivated the development of public key encryption using threshold decryption where multiple party members are assigned a unique share of the secret key which must be combined to be able to decrypt the ciphertext. This scheme is known as Threshold Public Key Encryption (ThPKE) [26].

ThPKE can be extended to FHE in the case where multiple parties each have a share of a secret key corresponding to the FHE public key. In this case, regular FHE is insufficient and calls for a distributed variant [27] referred to as Threshold Fully Homomorphic Encryption. Threshold FHE must allow for multiple parties to combine their shares of the secret key in order to decrypt the result of computations on the encrypted data in as general an access structure as possible. In order to implement such a scheme, the following is required for some access structure [27]:

1. A dealer must generate an FHE public key and corresponding shares of the

secret key that are distributed to all parties.

2. Given a ciphertext and a secret key share, any party must be able to compute their corresponding partial decryption.

3. Given this partial decryption for all parties, the plaintext message must be recoverable.

4. Given an adversary who corrupts a subset of the parties not included in the access structure, they should not be able to learn anything about the encrypted message even after seeing partial decryptions of other ciphertexts by honest parties.

Threshold FHE utilises the common t-out-of-n setting where a set of t+1 parties can combine their shares to decrypt the ciphertext while any set of t parties of fewer learns no information regarding the ciphertext [26]. Therefore threshold FHE allows for the computation on encrypted data followed by the decryption of the result by way of combining at least t+1 partial decryptions [26].

An example of an implementation of threshold FHE for smart contracts is the fhEVM framework which this project is going to be utilising to build the backrunning protocol. The fhEVM is an "encryption component" that encrypts everything under a single public key, which is generated using a threshold protocol that distributes differing parts of the global key among validators to further ensure security. This key is then used to encrypt inputs to protocols where computations can be performed on them locally by validators [25]. The fhEVM is an example of how FHE can be used on chain to add privacy and security to existing solutions such on-chain auctions, key-value databases, bridges and voting. Despite its many uses, FHE does provide concern for users. The main concern in the case of FHE is that its use on the blockchain has poor speed and performance, being "much slower than known alternatives" [25], with the overhead being approximately 10,000 times slower than the equivalent unencrypted computation [25].

2.3 Related Work

2.3.1 Dark Pools

An example of a dark pool can be seen in previous work published by ZAMA. The "fhe-darkPools"[1] github repository is an on chain implementation of a decentralised dark pool which uses the fhEVM framework to encrypt large trades in order to prevent "market panic and MEV" [28], using FHE to match the price to the amount between the buyer and the seller. This project uses homomorphic encryption in a similar way to achieve the backrunning of private transactions.

Another example of a decentralised dark pool is one produced by Sunscreen, which utilises threshold fully homomorphic encryption to minimise communication between parties [17]. Sunscreen first generates a key pair and distributes a portion

[1] `https://github.com/omurovec/fhe-darkpools`

of the private key to each committee member. Once a user submits an encrypted order, the committee members are then able to run the matching algorithm directly on the order, checking for possible matches [17]. Upon a successful match the committee members combine their respective portions of the private key which can then be used to decrypt the final transaction. Sunscreen splits the private key between the committee members to create a "distributed trust model" [17] ensuring that no one committee member can view the record book by simply using their own partial private key but rather needs the remaining partial keys from the rest of the committee members.

The Renegade dark pool is another example of an on-chain, decentralised dark pool which utilises SMPC for order matching and ZK-proofs for atomic settlement of matched orders. Renegade claims that this leads to minimal MEV and "higher-quality trade execution at midpoint prices" since no one can learn any information about a user's token balances, pending orders, or trade history [29]. Renegade uses an encrypted and distributed order book where matches between users' orders are inferred via a "cryptographically secure multi-party computation" [29]. Using this as well as zero knowledge proofs, Renegade attempts to solve the problems of non custodial trading such as MEV, pre-trade transparency ("non marketable trades that rest on a limit order book are visible to all third-parties, leading to quote fading"), post-trade transparency ("all third-parties can query the entire state history, allowing for tracking and tracing of trading activities") and address discrimination ("traders can see the origination address of all outstanding orders, leading to worse fills against sophisticated counterparties") [29].

2.3.2 Backrunning Private Transactions Using Multi-Party Computation

Recent research breakthroughs that motivated the development of this project were achieved in the 'Backrunning Private Transactions using Multi-Party Computation' by Flashbots [30]. Annessi first attempted to solve this problem utilising Intel's SGX since it is able to provide secure remote computation and thus "the correctness of an output of a trusted program can be assured even if it was executed on an untrusted machine" [30]. This therefore allows searchers to run the backrunning program on a trusted execution environment (TEE) within their own machines. Annessi further emphasised that all communication between the program and agents can be transport layer encrypted, mitigating the use of the enclave's output as an overt channel to disclose the private transaction [30]. Flashbots also go on to explain that they restrict the network communication from within the enclave in order to avoid any information leakage that can be used to reveal the transaction. This, however, also means that "connecting to the Ethereum P2P network is not possible" [30]. Two further covert channels where information can be leaked is through the use CPU usage and the allocation of memory pages or patterns of memory access, which can be mitigated by restricting the searcher's input in a way that does not affect or alter the program execution or reveals the transactions through covert channels [30]. However the need to tradeoff expressiveness and security has led Flashbots away from

the use of SGX and towards the use of MPC-Backed Backrunning.

MPC-Backed Backrunning "enables multiple parties to jointly compute a public function while keeping their input confidential, ensuring that only the output of the function is revealed." [30] Flashbots used a similar setup to the SGX program but instead of the backrunning algorithm being run on the searchers machine, it is now "executed as a communication protocol between the user and the searcher" [30]. The user inputs their transaction into the algorithm whilst the searcher inputs their searching strategy by way of their program in addition to inputting a secret key used to sign the output backrunning transaction. Based on these inputs the algorithm outputs a profitable backrunning transaction. Whilst demonstrating a successful private backrunning operation, this solution does present some challenges, namely the expressiveness and usability of the searcher language which is currently "low-level and prone to errors" [30]. It also poses problems with the practicality with regard to its agregious computational overhead, "rendering it impractical for real-world use." [30].

2.3.3 Leveraging Homomorphic Encryption for Maximally Extractable Value (MEV) Mitigation: Enabling Blind Arbitrage on Decentralised Exchanges

Moving on to research utilising FHE, in unpublished research, Flashbots utilised FHE in the form of the tfhe-rs Rust Library to build upon the previous challenges faced by the MPC-backrunning solution and to try and reduce the overhead in data transmission present in the MPC solution. To achieve this, Flashbots utilised levelled operation grouping, a form of graph-level optimisation to achieve significant performance gains [8]. Levelled operation grouping is performed in an attempt to minimise and reduce the levels of unnecessary operations requiring programmable bootstrapping (PB). Levelled operations are required when performing additions; they do not require bootstrapping and are not very computationally intensive. On the other hand, while PB is fast in TFHE relative to other schemes, it remains a performance bottleneck [8]. Despite this, these PB operations are necessary to allow for the computation of multiplications and divisions using lookup tables as well as to aid in containing the noise level within the acceptable range. Due to the difference in performance between the two types of operations, Flashbots decided to strategically merge together levelled operations in order to reduce the number of operations requiring PB and thereby reducing the overall amount of bootstrapping and thus the overall execution time [8].

In an attempt to further reduce the computational overhead, Flashbots utilised a reduced security parameter approach to achieve further performance gains [8]. Flashbots decided to reduce the FHE security parameter from 128 bits to 80 bits since long term confidentiality is not required within this context as a result of the inevitable publication of transactions once on chain. Flashbots further state that as a result of the performance gain achieved, combined with the "substantial protection against brute force attacks" [8], utilising the lower security parameter successfully struck a balance in the trade off between security and performance. This is especially

due to the new security parameter not compromising the integrity or correctness of the FHE scheme itself [8]. Despite the performance gain compared to previous solutions, this work by Flashbots is still limited by its abstracted nature as it can only handle the backrunning a single transaction at a time, significantly reducing the amount of MEV opportunities available to searchers. Another drawback of this solution is the restricted use case; being limited to one decentralised exchange restricts the searchers ability to apply diverse and novel strategies. To address these shortcomings, the following chapter presents the designed solution.

Chapter 3

Design

The previous chapter has set the context for the rest of the paper by describing and evaluating the background theories and relevant literature to decentralised finance, privacy enhancing technologies and relevant work.

This chapter that follows presents details on the design of the proposed protocol which aims to allow searchers to blindly backrun private user transactions through the use of the fhEVM framework. This solution aims to return a raw Ethereum transaction that, when executed immediately after the target transaction, leverages the price imbalance and extracts a profit.

Section 3.1 of this chapter details the motivations and aims behind some of the choices undertaken during the designing phase. These choices include the use of the fhEVM framework for encryption and the use of the public network to deploy the backrunning protocol.

Section 3.2 introduces the design of the system architecture including a high level overview and workflow of the system from decoding the user transaction to producing the profitable backrunning transaction.

Following this, section 3.3 of this chapter details the intricate design plans of the protocol including the decoding of transactions, the combining of multiple user transactions, the profit calculations and the creation of the backrunning transaction.

Section 3.4 of this chapter introduces some of the anticipated challenges faced during the development and implementation of this backrunning protocol. This section will also list the respective solutions for the outlined challenges.

This chapter then concludes in section 3.5 which introduces the steps taken to ensure the systems maintainability with regard to its robustness and scalability.

3.1 Design Choices

During the design and development of any protocol, it is essential to clearly outline the aims and motivations behind any undertaken decision. As such, this section explains the decisions and justifications behind the major design choices underpinning this study.

Fully Homomorphic Encryption Framework

The first critical design choice that was made during this design phase was the choice of encryption framework to be used. Within this protocol, this encryption framework is trusted to provide confidentiality through the use of FHE to ensure the privacy of both the user transaction and the searcher's strategy. Due to its importance within this solution, the encryption framework must strike an effective balance between security, performance and expressiveness.

In an attempt to achieve this balance, the fhEVM framework was selected over other frameworks as a result of a culmination of several factors. ZAMA's fhEVM framework "makes it possible to run confidential smart contracts on encrypted data, guaranteeing both confidentiality and composability" [31]. The fhEVM framework gives access to encrypted data types to allow developers to outline which states should be confidential, thus allowing for the encryption of private transaction data. One of the main factors contributing to this decision was that the fhEVM framework allows for the use of high precision encrypted integers (up to 64 bits of precision) [3] as well as the use of encrypted data types, giving developers the ability to mark which aspects of their smart contracts should remain private and which can remain public. In addition to this, ZAMA add to the usability of the fhEVM by allowing the use of all mathematical operations including addition, subtraction, multiplication, division and comparisons. This differs from other FHE frameworks which only provide the ability for addition and multiplication as can be seen in Figure 3.1. With regard to security, the fhEVM framework claims to provide "unprecedented levels of privacy and security" [3], achieved through the use of FHE to compute private states directly on chain, MPC to compute the threshold decryption of FHE ciphertexts and ZK proofs to ensure the integrity of encryption and decryption [3]. The fhEVM framework outperforms comparable frameworks in almost all aspects, as can be seen in Figure 3.1, and thus was entrusted as the FHE framework of choice.

Deployment

Another design choice that had to be considered during the design of this protocol was the deployment of the protocol. Be it on the searchers machine or on the public network as a smart contract, the deployment of the protocol has a significant effect on the user and searcher experience, in addition to the scalability of the solution.

On one hand, deploying the protocol on the searchers machine allows for a reduced computational load since the computations are being performed off chain rather than on chain and thus, are not limited by gas usage. In addition to this, running the backrunning protocol on the searcher's machine removes the need to encrypt the searcher's strategy. This reduced encryption within the protocol results in improved performance, especially when calculating profits.

On the other hand, deploying the backrunning protocol on the public network has the benefit of running the computations on the network's validators' machine rather than on the searcher's machine. This provides users with the increased transparency as a result of the public nature of the network. Additionally, computing on the validators' machines allows for a more consistent performance and compu-

	Zama fhEVM	Other FHE	ZK	Mixers	SGX
Operations supported	Everything	Additions & multiplications	AND & XOR	None	Everything
Privacy Model	Hides the data	Hides the data	Hides the data	Hides the identity	Hides the data
Data Availability	On-chain	On-chain	Off-chain	On-chain	On-chain
Encrypted state composability	Yes	Limited	No	No	Yes
On-chain PRNG	Yes	No	No	No	Yes
Developer Experience	Easy	Medium	Hard	Hard	Easy
Compliance	At the application level	At the user level	At the user level	At the user level	At the application level
Security	Proven secure	No security proof	Proven secure	Proven secure	Broken

Figure 3.1: fhEVM Comparison [3]

tational load since validators typically run on high performance, secure machines. However, a disadvantage of running on chain is that complex computations can often require more gas thus leading to higher transaction fees which in turn, reduce the profits extracted. Additionally, computing the backrunning protocol on the public network may yield scalability issues in the future due to the immutable nature of smart contracts. A more detailed plan for scalability and maintainability of this protocol is discussed in section 3.5.

As a result of the previously discussed advantages combined with the use of the fhEVM, this study deploys the backrunning protocol on the public network to be computed on the validators' machines.

3.2 System Architecture

The backrunning algorithm is very simple in theory. The searcher inputs multiple pending transactions in addition to their their searcher program, outlining their strategy for backrunning. A searcher's strategy is their plan for extracting value which typically consists of a range of parameters such as the market price on the exchange, the pricing function and the state of the pool [30]. These parameters can be utilised to calculate the amount of tokens that should be bought up to a certain price. Given these inputs, the output of this algorithm is either a successful, profitable backrunning transaction that leverages the market shifts created by the user's transactions to extract additional value or the output is an empty transaction in the case that a viable backrun was not possible given the searcher's strategy.

When there are multiple user transactions, they must first be combined to see their net effect on this exchange. This "collective" transaction is then vetted for compliance with the searchers requirements to evaluate its efficacy in being back-

run. Following this, the searchers calculations and validity checks are applied to the transaction in order to derive the amount of tokens needed to trade in order to effectively exploit the arbitrage opportunity. At which point, the optimal backrunning parameters are calculated and combined to create the backrunning transaction offering maximum profitability whilst also complying with the searchers acceptance criteria. The use of FHE here is critical to ensure that the users transactions and the searchers algorithm remain private at all times, whilst allowing the searcher to successfully and efficiently apply their program. This is especially crucial since a key difference between this protocol and the Flashbots solutions discussed in the previous chapter is that this protocol is deployed on the public network and not the searcher's machine meaning that it is not sufficient to leave the searchers strategy unencrypted.

Given that these transaction are to be executed sequentially as part of a block, the target transaction is the last of the user transactions to be executed and thus is the one that creates the arbitraging opportunity to be exploited. In order to derive the most profitable target transaction, this protocol compares their relative trade sizes by iterating over each transaction and isolating a unique transaction at every iteration to act as an interim target transaction. At each iteration, the remaining, non-target, transactions are applied to the decentralised exchange by trading in their tokens and using the exchange's ruling equation ($x \cdot y = k$) [15] to calculate the updated token reserves. Given both the interim target transaction and the update reserves, the relative trade size of the interim target transaction can be calculated to infer the resultant price imbalances. The interim transaction with the largest relative trade size is then chosen as the target transaction to be backrun. It's important to note that throughout the program, a boolean compliance flag is always set to True and is only set to False once a transaction does not pass the acceptance or profitability criteria. Upon passing the acceptance criteria, the required searcher's calculations and validity checks are performed on the transaction with the highest relative trade size to infer the potential profit and the amount to trade to receive said profit. Given these calculation, the protocol uses the optimal parameters to build a profitable backrunning transaction that exploits the price imbalances and extracts value from them. In the event that the transaction does not pass the acceptance or profitability criteria, an empty transaction will be returned to the searcher. It is important to note, that due the combination of multiple transaction being challenging from a computational point of view, the number of transactions that are able to be combined are limited to an amount that is computationally feasible.

This solution is to be implemented using the following four smart contracts:

- **Main.sol**
  This smart contract acts as a facade, utilising known design patterns [32] to provide a simplified interface that allows the searcher to easily interact with the protocol. This contract directly interacts with the other three contracts in order to provide a simple and usable interface for the user.

- **Backrun.sol**
  This contract acts as the complex subsystem for which the previous contract is the facade for. This contract implements the main functions of the solution

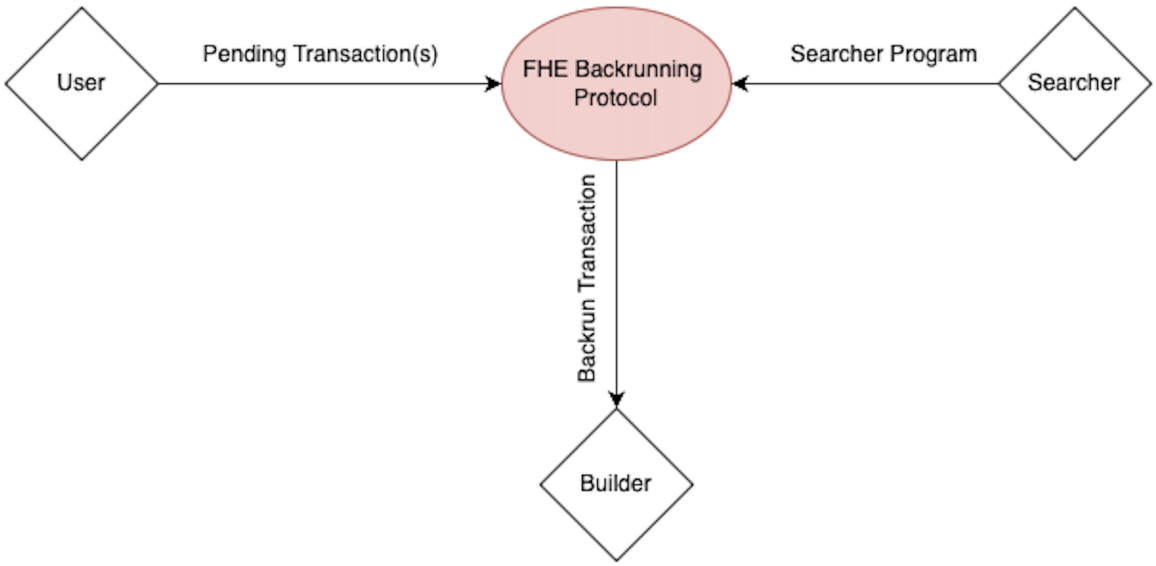

Figure 3.2: Backrunning Protocol

including updating the token quantities, calculating the amount to trade and calculating the searcher's potential profit.

- **RLPCoder.sol**
  This contract performs the RLP decoding and encoding of transactions. This contract is responsible for the decoding of the initial user transactions and the encoding of the final, backrunning transaction.

- **MultipleTx.sol**
  This contract implements the algorithm for the combination of multiple transactions to allow searchers to implement strategies of a greater complexity.

The following section of the report delves into the design of this solution in greater detail. Additionally, Figure 3.2 showcases the high level workflow of the backrunning protocol while Figure 3.3 showcase a flowchart of the main.sol smart contract.

3.3 System Design

To implement this high-level design, a more detailed, low level design is necessitated. This section implements this by first detailing the design of the transaction decoding algorithm in section 3.3.1 followed by the design of the transaction combination in section 3.3.2. After this, this section moves to detail the search strategy in section 3.3.3 before describing the design of the profit calculation in section 3.3.4. Finally, section 3.3.5 details the building of the backrunning transaction.

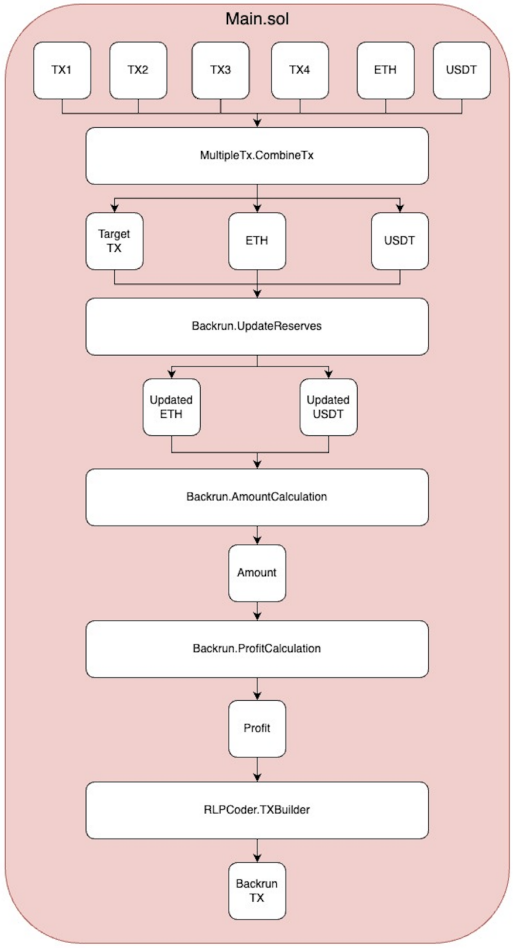

Figure 3.3: Flowchart of the main.sol smart contract

3.3.1 Transaction Decoding

The first step in designing the backrunning protocol is the decoding of raw transactions into their constituent parts. Ethereum transactions are recursive length prefix (RLP) encoded, and therefore in order to extract the necessary information, decoding the transaction is the necessary first step. The strategy for this part of the protocol is to begin by creating an unencrypted version of the code and then follow this up with the integration of the fhEVM framework to privatise and secure the transaction's fields.

Raw Ethereum transactions typically consist of the following fields:

- **Nonce:** An index that represents the total number of transactions sent from an address.
- **Gas Price:** The cost per unit of gas spent for the transaction, in Ether and Gwei [1].
- **Gas Limit:** The maximum amount of gas allocated for the transaction [1].
- **To:** The receiving party of the transaction [1].

- **Value:** The value being transacted in Ether and fiat value [1].
- **Data:** Additional data included for the transaction [1].
- **From:** The sending party of the transaction [1].
- **v, r and s:** ECDSA signature components.

In order to decode the transaction, the data type of the encoded bytes must first be deciphered, which is done by utilising the RLP rules and checking which range the first byte of data belongs to. If the byte falls within the (0x00 - 0x7f) range then the byte is a string and should be decoded as it is. Similarly, if the byte lies within (0x80 - 0xb7) or (0xb8 - 0xbf) ranges then this represents a string with the former range indicating a short string and the latter a long string. While the (0xc0 - 0xf7) and (0xf8 - 0xff) ranges represent short and long lists respectively [33, 34]. After deciphering the data type, the next step in decoding the transaction is retrieving the length of the byte array. The byte length can be deciphered by subtracting the first byte of the array from the first byte of its data type range [33, 34]. By repeating these steps for every byte of the transaction it can be decoded into the constituent fields.

3.3.2 Combining Multiple Transactions

The second, and most complex, step in designing the backrunning solution is the efficient combination of multiple transactions to find the most profitable order of transactions. Despite appearing simple, executing this problem in a efficient manner is far from trivial especially given the nature of Ethereum and its computational constraints. As a result, designing and implementing a computationally viable solution to this problem is bound to be the crux of this backrunning protocol where the inherent difficulty of this problem stems from the fact that the search space increases factorially with the addition of every transaction. This, combined with the use of encrypted operations throughout creates a complex, gas intensive problem needing of an efficient solution. Essentially an optimisation problem, this step of the backrunning protocol requires an optimal combination of transactions that maximises profit whilst minimising gas costs. Thus, this section of the report outlines two possible solutions: the former being comprehensive but inefficient and the latter being efficient but incomprehensive.

Brute Force Combination

The first of these solutions to combine multiple transactions comprehensively evaluates the profitability of all transaction combinations. This brute force approach follows the workflow shown in Figure 3.4 which ensures that the most optimal and most profitable combination of transactions is always returned to the user.

Implementing this first requires the updating of the pool reserves according to the pools ruling function $x \cdot y = k$ [15]. This protocol should utilise this function to apply the trades on the pool for all transactions other than the potential target

transaction, eventually returning the updated pool reserves. Given this updated pool reserves and the target transaction, the protocol should move to calculate and log the profitability of all combinations. The most profitable of which being returned to the searcher and utilised in the remaining methods of the backrunning solution.

This solution offers a comprehensive result, ensuring that the protocol will always return the most profitable combination of transactions, regardless of the number of transactions offered. This solution, however, requires a run through of the entire backrunning protocol for every transaction, with each run decoding and calculating the token reserves. The recursive nature of this solution will undoubtedly result in a complex and computationally expensive solution that necessitates another, more economical solution.

Combination Based on Relative Trade Size

Alternatively, another method to combine transactions diverges away from the brute force nature of the previous solution and leverages the ruling function of Uniswap to individually analyse all of the transactions. This method takes advantage of the fact that larger trades result in greater price shifts and thus a greater opportunity to successfully backrun the transaction. Therefore, this proposed solution compares the relative trade sizes of all transactions and returns the transaction with the largest relative trade as the backrunning target transaction due to the increased likelihood of profitability.

Similar to the previous solution, the protocol must first update the token reserves by applying all but one of the trades on the pool using the pool's ruling equation. Based on these resulting reserves the protocol must then calculate the relative trade size of each transaction according to Equation 3.1 by initially decoding and extracting the amount traded in from the target transaction. This process is repeated for all potential target transactions to produce their respective relative trade sizes. Given these, the transaction with highest relative trade size and the updated pool reserves are returned to be used in the remainder of the protocol. This solution reduces the problem from a multi-transaction backrun to a single transaction backrun by aiming to estimate the profitability of each transaction based on the amount traded in rather than directly evaluating the profits. This therefore reduces the complexity of the solution greatly especially compared to the previous solution.

$$relativeTradeSize = \frac{amountIn}{tokenReserve} \tag{3.1}$$

This solution, attempts to overcome the factorial complexity of the first solution by analysing the trades intuitively rather than explicitly. As a result, this allows it to significantly reduce the computational overhead needed to evaluate all of the user transactions passed in. A possible downfall of this solution, however, is that since it is not explicitly evaluating the profitability of all transactions and is going off of the basis that larger trades result in larger price shifts and thus greater opportunity, it is

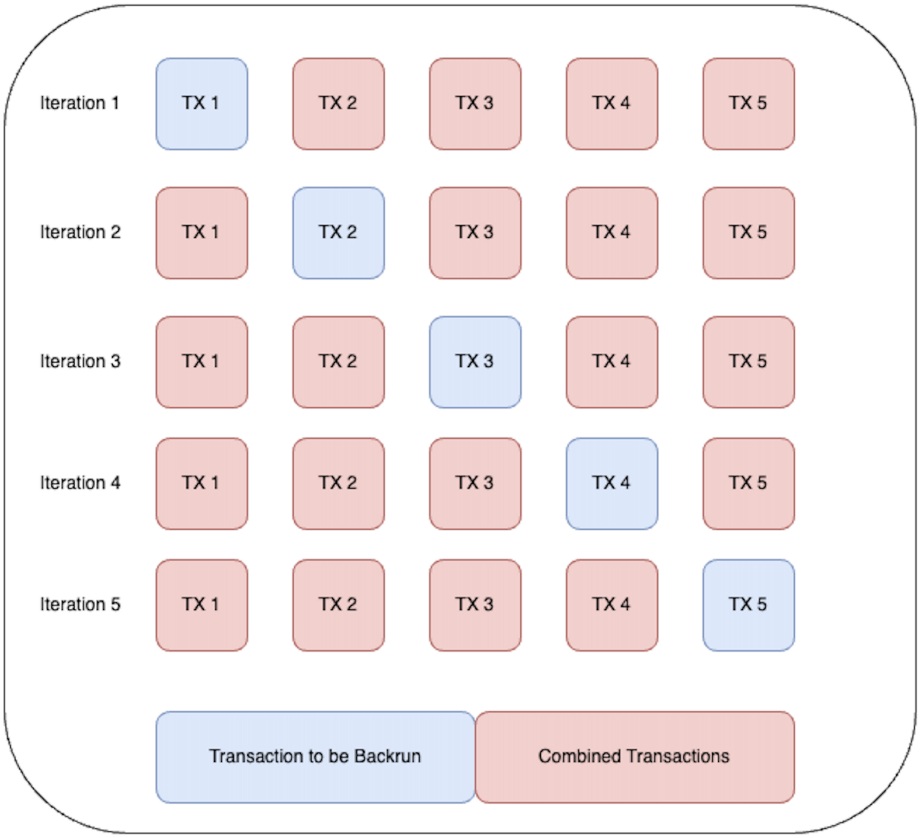

Figure 3.4: Transaction Combination

entirely possible that some more profitable combinations may be missed due having a lower relative trade size, however unlikely this may be.

This step in designing the protocol is essential as it gives searchers the ability to backrun multiple transactions. As a result, this step allows for the creation of more complex strategies that can leverage the market's price imbalances for profit.

3.3.3 Searcher Strategy

Inputting the searchers strategy is the third step of designing the protocol. Within this protocol the searchers strategy are the constants that are needed for computing the profit. These constants include the pool's reserves, the maximum buy price of the token, the minimum sell price of the token, the searcher's nonce and the searcher's address.

These searcher provided constants, the decoded transaction parameters and some additional hard coded constants are all necessary to calculate the profit for the back-running transaction. These fixed constants include the:

- **Fee Dividend - (3)**
  - The numerator used to calculate the UniswapV2 fee.
- **Fee Divisor - (1000)**

  - The denominator used to calculate the UniswapV2 fee.

- **Token Precision - (1000000)**
  - The number of decimals used in the protocol.
- **Constants to Calculate the Amount In - (2 & 4)**
  - The constants used to calculate the profits in equation 3.5.
- **Cost of Arbitrage - (2)**
  - The cost of executing both the on-chain transaction and the trade on the exchange.
- **minimum profit - (0)**
  - The minimum amount of profit that the backrunning transaction should make.

For the sake of efficiency, these parameters are hard coded into the contract in order to minimise the gas usage. On the other hand the other set of parameters are not hard coded and are either derived from the user transaction or input directly by the searcher.

3.3.4 Profit Calculation

The next step of the protocol utilises these parameters and constants to calculate the amount of tokens that are needed to be bought in order to create a profitable backrunning transaction. The equation used to calculate the amount to backrun stems from the ruling equation of the UniswapV2 exchange ($x \cdot y = k$) [15]. This ruling equation is first used to calculate the quantities of the tokens post trade as seen in equations 3.2 and 3.3:

$$x = x_{preSwap} + amountTraded \tag{3.2}$$

$$y = \frac{x_{preSwap} \cdot y_{preSwap}}{x_{preSwap} + amountTraded \cdot (1 - fee)} \tag{3.3}$$

These equations can then be used to compute the price of the trade as seen in equation 3.4:

$$price = \frac{x \cdot y}{x + amount \cdot (1 - fee)} : x + amountTraded \tag{3.4}$$

Equations 3.2, 3.3 and 3.4 can subsequently be transformed to express the amounts of tokens to be sold. The resulting transformation can be seen in equation 3.5:

$$amount = \frac{\sqrt{x \cdot prec \cdot (fee^2 \cdot x \cdot prec + 4 \cdot price \cdot y \cdot (1 - fee))} + x \cdot prec \cdot (fee - 2)}{2 \cdot prec \cdot (1 - fee)} \tag{3.5}$$

Within equation 3.5, the $prec$ variable references the token precision which is multiplied with the x variable. Furthermore, the fee variable in Equation 3.5 references the Uniswap fee of 0.3% while the $price$ variable represents the searcher's maximum buy price.

Upon calculating the amount to be backrun, the protocol then moves to calculating the profitability of the backrunning transaction. Equation 3.6 is used to calculate this profitability by combining the searcher constants of $sellPrice$, $buyPrice$ and $costOfArbitrage$ with the $amountOut$ variable which is the quantity of the y token received by the searcher when trading in the amount calculated in equation 3.5.

$$Profit = \frac{(sellPrice - buyPrice) \cdot amountOut}{prec} - costOfArbitrage \tag{3.6}$$

Validity Criteria

Upon calculating the profitability, the protocol performs several comparisons on both the original user transaction as well as the calculated amount and profit to ensure that the transaction being sent to the builder is both valid and profitable. With regard to the user transactions, the first comparison that the protocol should perform is to ensure that the destination address of the transaction is the UniswapV2 router. Following this, the protocol should then guarantee that the deadline of the backrunning transaction is greater than or equal to the outlined minimum deadline. Finally, the protocol should then make sure that the path length of the data is equal to two. Upon calculating the searcher amount and profit, the protocol should ensure that both of these are greater than 0. Only once all of these comparisons hold does the protocol create and return the backrunning transaction to the searcher. In the case that the acceptance criteria is not met, the protocol should return an empty transaction to the searcher.

3.3.5 Creating The Backrunning Transaction

The final step of deigning our protocol is to create the final backrunning transaction using some of the initial parameters as well as the calculated amount to be backrun. The final transaction consists of the following fields:

- **Nonce**
- **Gas Limit**
- **Gas Price**
- **To Address** (The UniswapV2 router)

- **Value**
- **Data: MethodID**
- **Data: Amount In** (The amount calculated using equation 3.5)
- **Data: Amount Out**
- **Data: Address Offset**
- **Data: To Address**
- **Data: Deadline**
- **Data: Address[] length**
- **Data: Token 1**
- **Data: Token 2**

With regard to the $MethodID$ field, its value in the backrunning transaction is the opposite of the original user transaction. Similarly the values of the $Token1$ and $Token2$ fields are also switched compared to the values of the user transaction.

Following the construction of this transaction, it must be then RLP encoded and returned to the searcher alongside the initial user transactions so that they can be sent to the block builder. In order to RLP encode the transaction, utilising the $solidity - rlp - encode$ library[1] [35] by Bakaoh. This library consists of functions to encode all data types including integers, bytes, lists and strings. To encode the transaction, the protocol must first create an array of the required transaction fields. This array consists of the nonce, gas price, gas limit, to address, value and the data field. Each of these fields must be RLP encoded prior to being added to the array, according to their respective data type. With regard to the data field, it consists of additional fields and it's fields are unique to the specific method being called. To encode this, the concatenation of the method ID and the application binary interface (ABI) encodings of the method's specific fields are RLP encoded and appended to the array alongside the previous fields. This array is then RLP encoded using the "encodeLists" function to create the complete, encoded transaction which is then returned to the searcher.

3.4 Challenges of The System

When designing the proposed solution it is vital to address some of the anticipated challenges that are common within the field of decentralised finance. This section of the report moves to outline some of these challenges in addition to their respected solutions.

[1] `https://github.com/bakaoh/solidity-rlp-encode/tree/master`

Challenge 1: The main challenge that may be faced in this protocol is the integer precision in the fhEVM framework. With a maximum integer precision of 64 bits, this makes some calculations difficult due to Ethereum's 18 digit precision. This results in an increased likelihood of an overflow when calculating the trade in size or the searcher's profits.

Solution: A solution to this would be to limit Ethereum's precision to 6 bits to match USDT rather than increasing USDT's precision to 18. Despite lowering the total precision and accuracy of the calculation, this solution allows the protocol to effectively perform the calculation's while reducing the likelihood of an integer overflow.

Challenge 2: The second challenge of this protocol is the combining of multiple transactions in a computationally feasible way.

Solution: A solution to this would be to limit the amount of transactions that the searcher is able to backrun in order to ensure the efficient execution of the protocol. In addition to this, the protocol will return the best combination of transactions by evaluating the relative trade sizes of each target transaction rather than brute force searching all of the transactions to check their profitability.

Challenge 3: Another challenge of this protocol comes with the calculation of the amount to be traded which requires performing division by a scalar. This is difficult in Solidity due to the lack of representation for floating point integers.

Solution: The solution to this challenge is to utilise homomorphic division through the use of the "TFHE.DIV" function from fhEVM. However it is important to note that this function does not yet support an encrypted divisor and therefore, any FHE encrypted dividend can only be divided by a plaintext divisor.

Challenge 4: Similar to the previous challenge, evaluating a square root is another operation that is required when calculating the amount to be traded. This is similarly difficult in Solidity as a result of the lack of floating point integer representation.

Solution: In order to evaluate the square root, this project will be utilising the Newton method which utilises homomorphic addition to approximate the square root.

Challenge 5: Retrieving the up to date information about the token quantities in the pool is a another challenge that must be addressed in this protocol.

Solution: In order to effectively retrieve accurate, current information about the pool, the searcher must input the current state of the pool as part of their

strategy. This is done in order to not increase the computational load by implementing an oracle that constantly retrieves information regarding the state of the pool.

3.5 System Maintainability

A projects maintainability is determined by comparing the effort spent maintaining the existing code against the effort spent writing new code. if the former consumes greater effort, this implies poor maintainability where modifications and corrections to the code are needed to skew the comparison towards the latter. The product quality model defined in ISO/IEC 25010 defines modularity as the "degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements."[36]. Modularity is described as comprising of [36]:

- **Modularity:** "The Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components."
- **Reusability:** "The Degree to which a product can be used as an asset in more than one system, or in building other assets."
- **Analysability:** "The Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified."
- **Modifiability:** "The Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality." [36]
- **Replaceability:** "The Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met."

This proposed system attempts to adhere to the above characteristics in order to minimise the efforts spent on maintaining the code and increase the efforts spent on writing new code. This focus on maintainability ensures long term robustness and scalability which is vital in the wider context, given the ever changing requirements and advancements of Ethereum.

Chapter 4

Implementation

This chapter describes the steps taken in implementing the proposed solution whose design was expounded in the preceding chapter. Section 4.1 begins by showcasing the implementation of the transaction decoding smart contract and its development from the previous designs. Section 4.2 then details the implementation and adjustment of the methods utilised to combine multiple transactions to allow for complex searcher strategies. Following this, section 4.3 of this study addresses the next step of this proposed solution by showcasing the steps taken to calculate the amount of tokens to trade in as well as the searcher's expected profit. Finally, this chapter concludes by showcasing the aggregation of parameters and their encoding, as required to create a successful backrunning transaction.

Throughout this chapter all encrypted variables are coloured in blue while all unencrypted variables remain as black. An overview of the proceeding algorithms and how they connect to each other can be seen Figure 3.3.

4.1 Transaction Decoding

4.1.1 Manual Decoding

The initial implementation of the transaction follows on from the previously outlined design plans by firstly decoding the length of the transaction by subtracting the first byte of the array from the first byte of its data type range. Then, given the length, the protocol iterates over the fields to decode each field individually.

Algorithm 1 showcases the decode function which takes an RLP encoded user transaction and returns an array of decoded fields. The algorithm begins by decoding the length of the user transaction by calling Algorithm 2 which first requires the length of the user transaction to be greater than 0 to ensure that the input is not null as well as requiring that the first byte of the user transaction is greater than 0xc0 to ensure that the user transaction is indeed a list and therefore contains the necessary fields. In the case of a short list, the algorithm returns an offset of 1 and the length of the list by subtracting the the first byte of the array from the first byte of its data type range (0xc0). In the case of a long list, the length of the list is deciphered similarly to short lists, but to get an accurate length of the embedded list a helper function

$ArraytoInteger$ is utilised to convert the list into an integer. Given these, Algorithm 2 returns both the offset as well as the length of the data.

Given the length of the data, Algorithm 1 moves to iterate over all of the fields. At each iteration Algorithm 3 is called to individually decode the transaction's fields. This algorithm begins by setting the prefix of the user transaction as first byte of the field to aid in the calculation of its length. Following this, the algorithm first addresses the case by decoding a field as it is, by returning the corresponding byte as well as its length (1). Then, the algorithm moves to decode short strings by returning the corresponding slice of the user transaction as well as the length of the string. Long strings are decoded similarly by returning the appropriate slice as well as the length of the string. In the case that the prefix of the field does not belong to one of these cases, the function reverts. Given the decoded field, Algorithm 1 iterates over every field by updating the offset with the length of the previous field. Upon completion of every iteration, the algorithm returns the $decodedFields$ array.

Despite successfully and effectively decoding the RLP encoded user transaction, this implementation was computationally expensive even before the integration of encrypted operations, which would have only increased gas consumption. As a result, this motivated the development of an updated solution using a third party library.

Algorithm 1 Decoding Algorithm

function DECODE(userTransaction)
  $offset, dataLength \leftarrow decodeLength(userTransaction)$
  for $i < dataLength$ **do** ▷ Iterates over all fields
    $length \leftarrow 0$
    $decodedFields[i], length \leftarrow decodeItem(userTransaction, offset)$
    $offset \mathrel{+}= length$
    $i\,{+}{+}$
  end for
  return $decodedFields$
end function

4.1.2 Automated Decoding

This updated implementation of the transaction decoding algorithm uses a Solidity library for Ethereum's RLP decoding (Solidity-RLP)[1] [37]. This library first converts the encoded bytes to an internal data structure ($RLPItem$) through the $toRlpItem(Bytes)$ function [37]. Following this, the library can decode this data structure into the desired data types including lists, bytes, addresses, uint's or booleans. This destructuring is performed using the built in functions in the library such as the $toUint()$ and the $toAddress()$ functions.

This library, in combination with the fhEVM framework was used to RLP decode the input transaction as well encrypt the individual fields. The fhEVM functions

[1] `https://github.com/hamdiallam/Solidity-RLP`

Algorithm 2 Length Decoding Algorithm

Require: $userTransaction.length > 0$
Require: $userTransaction[0] \geq 0xc0$ ▷ User transaction must be a list
function DECODELENGTH(userTransaction)
$prefix \leftarrow userTransaction[0]$
$length \leftarrow userTransaction.length$
if $prefix \leq 0xff \ \& \ length > prefix - 0xbf$ **then** ▷ Short Lists
return$(1, prefix - 0xc0)$
else if $prefix \leq 0xff \ \& \ length > prefix - 0xf7$ **then** ▷ Long Lists
$listLength \leftarrow prefix - 0xf7$
$dataLength \leftarrow ArrayToInteger(userTransaction[1 : listLength])$
return$(1 + listLength, dataLength)$
else
$Revert$
end if
end function

Algorithm 3 Item Decoding Algorithm

function DECODEITEM(userTransaction, offset)
$prefix \leftarrow userTransaction[offset]$
if $prefix \leq 0x7f$ **then** ▷ Decode as is
return$(userTransaction[offset], 1)$
else if $prefix \leq 0xb7$ **then** ▷ Decode short string
$length \leftarrow prefix - 0x80$
$start \leftarrow offset + 1$
return$(userTransaction[start : start + length], length + 1)$
else if $prefix \leq 0xbf$ **then** ▷ Decode long string
$listLength \leftarrow prefix - 0xb9$
$dataLength \leftarrow ArrayToInteger(userTransaction[offset + 1 : listLength])$
return$(userTransaction[offset+listLength+1 : dataLength], listLength+$
$dataLength + 1)$
else
$Revert$
end if
end function

of $asEuint64$ and $asEaddress$ were used to encrypt the decoded transaction fields and convert their data types to their respective encrypted versions (euint64 and eaddress). These functions convert the decoded transaction fields into unique FHE ciphertexts thereby privatising the user's transactions.

Algorithm 4 aims to decode the relevant fields of user transactions and privatise the confidential fields by way of the previously mentioned fhEVM functions. To achieve this, algorithm 4 begins by first decoding the nonce, gas price, gas limit, to address and value fields, which are first extracted by converting the user transaction to the internal $RLPItem$ data structure. Following this, the user transaction was destructured to a list using the $toList()$ function whereby each index of the list represents a unique field in the transaction. Having identified and isolated the individual fields, the $toUint()$ and $toAddress()$ functions were used to decode the respective fields. Given these decoded fields, the appropriate fhEVM functions were used to encrypt the private fields including value field and elements of the data field. Thus resulting in the encrypted, decoded transaction.

With regard to the data field, since it encapsulates other method related fields, the algorithm first decodes and extracts the first byte which contains the method ID. Depending on the value of this, the data field is decoded and encrypted differently using the $decodeTokensForEth$ and the $decodeEthForTokens$ functions. These functions are decoded differently since they each encapsulate different parameters and thus necessitates the use of different functions to decode them. The $decodeTokensForEth$ function decodes the following fields: amount in, minimum amount out, address offset, to address, deadline, address length, token 1, token 2 while the function $decodeEthForTokens$ does not include the amount In field given that Ethereum is being traded in. It is also important to note that unlike previous fields, the data field is also ABI encoded and thus requires further ABI decoding before it can be encrypted. Within the data field, both the amount in and minimum amount out fields are converted to FHE ciphertexts to privatise the size of the user's trade.

It is important to note that in order to reduce the computational load of the solution, only private or sensitive parameters are encrypted. These includes the value, amount in and amount out parameters, which must be encrypted to ensure the protection and privatisation of the user's trades. On the other hand, parameters such as the destination address, path length and deadline are left unencrypted due to their use in confirming the validity of the user transaction. The details of validity checks are discussed in greater detail in section 4.3.3.

4.2 Combining Multiple Transactions

A crucial step in this carried study is allowing the searcher to create more complex strategies by allowing for the combination of multiple transactions. As outlined in the previous chapter, this step in the protocol is far from trivial, especially with regard the computational complexity.

When designing this step, two differing solutions were attempted with the aim of analysing then deriving the optimal design at the implementation stage. This

Algorithm 4 Transaction Decoding Algorithm

function DECODETX(userTransaction)
  $nonce \leftarrow userTransaction.toRlpItem().toList()[0].toUint()$
  $gasPrice \leftarrow userTransaction.toRlpItem().toList()[1].toUint()$
  $gasLimit \leftarrow userTransaction.toRlpItem().toList()[2].toUint()$
  $to \leftarrow userTransaction.toRlpItem().toList()[3].toAddress()$
  $value \leftarrow TFHE.asEuint64(userTransaction.toRlpItem().toList()[4].toUint())$
  $data \leftarrow userTransaction.toRlpItem().toList()[5].toBytes()$
  $methodID \leftarrow data[0:4]$
  if $methodID$ is $SwapExactTokensForETH$ **then**
    $decodedData \leftarrow decodeTokensForEth(data)$
  else if $methodID$ is $SwapExactEthForTokens$ **then**
    $decodedData \leftarrow decodeEthForTokens(data)$
  else
    revert
  end if
  return$(nonce, gasPrice, gasLimit, to, value, decodedData)$
end function

section reports on the implementation of the brute force combination, followed by the relative trade size combination.

4.2.1 Brute Force Combination

The first method implemented to find the optimal combination of transactions was a brute force combination that iterates over all possible combination of transactions, calculating their profits at each iteration, and eventually returning the most profitable iteration to the searcher.

Algorithm 5 showcases the implementation of the brute force algorithm. This algorithm begins by assigning the transactions to an array in order for them to be iterated over. At each iteration of a possible target transaction, all other transactions are iterated over and the cumulative sums of their respective token is calculated. Given the sums of each token being traded in, the new respective reserves are calculated using the $updateReserves$ function which takes both the input amounts as well as the current reserves. This is repeated twice to gain the final reserves after both tokens have been taken into account. Given the potential target transaction and the updated reserves, the profit of the transaction is calculated using equations 3.5 and 3.6. The details and the implementation of the amount and profit calculation is covered in greater detail in the following section. Following this, the profit of this target transaction is compared with the previous values and updates the parameters of the transaction in the event that this combination of transactions leads to the highest profit. Eventually, returning the target transactions as well as its respective updated reserves to the searcher after iterating over every combination of transactions.

A feature of Algorithm 5 is that its explicit calculation of profits, guarantees that the most profitable combination is always returned. However, it is hampered by its

complexity. This algorithm performs the $calcProfit$ calculation for every transaction and the $updateReserve$ calculation factorially for every transaction. This leads to an extremely high and unworkable solution that demands another faster, more efficient solution.

Within this algorithm, the amount of tokens traded in by the user is FHE encrypted to ensure the privacy of the user's trade. Similarly the sums of the trades stored in the $usdtSum$ and $ethSum$ variables are also encrypted. These encrypted values within the fhEVM are wrappers around handles (h) which are represented as $uint256$ values [6]. These handles are computed by applying a cryptographically secure hash function to the FHE ciphertext (c). ($h := Keccak256(c)$) [6]. When performing operations on ciphertexts the fhEVM does this via the handle only and do not explicitly control their bytes [6].

Algorithm 5 Transaction Combination Using Brute Force Algorithm

function COMBINETX(tx1, tx2, tx3, tx4, ethReserve, usdtReserve)
  $tx[0] \leftarrow tx1$
  $tx[1] \leftarrow tx2$
  $tx[2] \leftarrow tx3$
  $tx[3] \leftarrow tx4$
  for $i < 4$ **do**
    $(targetAmountIn, targetMethodID) \leftarrow getAmountIn(txs[i])$
    for $j < 4$ **do**
      if $i \neq j$ **then**
        $(amountIn, methodID) \leftarrow getAmountIn(txs[j])$
        if $methodID$ is $swapExactTokensForEth$ **then**
          $usdtSum \leftarrow usdtSum + amountIn$
        else
          $ethSum \leftarrow ethSum + amountIn$
        end if
      end if
    end for
    $(ethReserve, usdtReserve) \leftarrow updateReserves(ethReserve, usdtReserve, usdtSum)$
    $(ethReserve, usdtReserve) \leftarrow updateReserves(ethReserve, usdtReserve, ethSum)$
    $profit \leftarrow calcProfit(targetAmountIn, ethReserve, usdtReserve)$
    if $profit > maxProfit$ **then**
      $maxProfit \leftarrow profit$
      $maxIndex \leftarrow i$
      $maxEth \leftarrow ethReserve$
      $maxUSDT \leftarrow usdtReserve$
    end if
  end for
  return$(tx[maxIndex], maxEth, maxUSDT)$
end function

4.2.2 Combination Based on Trade Size

In order to address the issues of the previous solution, a solution that does not explicitly calculate profitability but rather exploits the market's dynamics to predict the most optimal combination was attempted.

Similar to the brute force solution, algorithm 6 begins by iterating over all possible target transactions, extracting their trade amount and method ID at each iteration. Following this, the sum of the trades of all non-target transactions, for that iteration, is collected and used to updated the token reserves. This method of updating the reserves was prioritised over incrementally updating the reserves for each individual transaction since this method minimises the calls to the $updateReserves$ function and ultimately reduces the gas cost and general overhead of the protocol greatly. Following this, the relative trade size is calculate by dividing the target transaction's trade in size by the previously calculated updated reserves. This is done since this function is used to calculate the optimal ordering of transactions, and as such the target transaction is the final transaction to be executed on the pool before the searcher's transaction and thus is the target of the backrun. The calculation of the ratios is based on the market dynamics, whereby the larger the relative trade size, the larger the price shift and thus the greater the opportunity for arbitrage and value extraction. Following the iteration over all transactions, the most profitable transaction is then returned to the searcher alongside the respective ETH and USDT reserves.

Within this algorithm the user's trade in size is FHE encrypted, using the $THFE.asEuint64$ function. This encryption of trade size is done regardless of whether the transaction is the potential target of the backrun or not. Ideally both sums of the non-target trade sizes would also be encrypted. However due to the 10,000,000 gas limit imposed by the ZAMA devnet, having 16 encrypted additions was not feasible.

While this method greatly outperforms Algorithm 5 with regard to gas consumption and efficiency, it is not as comprehensive as the previous solution due to the lack of explicit profit calculation. Despite this, when comparing both solutions they both returned the same answers in all of performed tests which justifies the use of Algorithm 6 despite its potential infrequent inconsistencies.

4.3 Searcher Strategy and Profit Calculation

Following the combination of multiple transactions, the necessary next step in our protocol is to calculate the amount of tokens that the searcher must trade in to effectively backrun the transactions and extract a profit. There are the four main phases required to achieve this:

1. Updating the token quantities.
2. Calculating the amount to trade in.
3. Calculating the searcher's profits.
4. Assessing the validity of transaction.

Algorithm 6 Transaction Combination Using Relative Trade Size Algorithm

function COMBINETX(tx1, tx2, tx3, tx4, ethReserve, usdtReserve)
  $tx[0] \leftarrow tx1$
  $tx[1] \leftarrow tx2$
  $tx[2] \leftarrow tx3$
  $tx[3] \leftarrow tx4$
  for $i < 4$ **do**
    $(targetAmountIn, targetMethodID) \leftarrow getAmountIn(txs[i])$
    for $j < 4$ **do**
      if $i \neq j$ **then**
        $(amountIn, methodID) \leftarrow getAmountIn(txs[j])$
        if $methodID$ is $swapExactTokensForEth$ **then**
          $usdtSum \leftarrow usdtSum + amountIn$
        else
          $ethSum \leftarrow ethSum + amountIn$
        end if
      end if
    end for
    $(ethReserve, usdtReserve) \leftarrow updateReserves(ethReserve, usdtReserve, usdtSum)$
    $(ethReserve, usdtReserve) \leftarrow updateReserves(ethReserve, usdtReserve, ethSum)$
    if $targetMethodID$ is $swapExactTokensForEth$ **then**
      $ratio \leftarrow \frac{targetAmountIn}{usdtReserves}$
    else
      $ratio \leftarrow \frac{targetAmountIn}{ethReserve}$
    end if
    if $ratio > maxRatio$ **then**
      $maxRatio \leftarrow ratio$
      $maxIndex \leftarrow i$
      $maxEth \leftarrow ethReserve$
      $maxUSDT \leftarrow usdtReserve$
    end if
  end for
  return$(tx[maxIndex], maxEth, maxUSDT)$
end function

4.3.1 Updating Token Quantities

Once transactions have been combined, the searcher's protocol must then calculate the effects of the target transaction on the liquidity pools. As mentioned in the previous chapter, the UniswapV2 governing function was utilised to update the token quantities post user trade.

Algorithm 7 showcases the calculation of the pools' liquidity depending on which UniswapV2 function was invoked by the user. This algorithm begins by calculating the fee owed to Uniswap. This 0.3% fee is calculated from the amount being traded in despite actually being taken from the amount traded out. This fee is then subtracted from the amount being traded in, at which point this result is used to calculate the updated liquidity for the output token. As for the liquidity of the token being traded in, this is calculated simply by adding the amount traded in to the pre trade quantities. This algorithm returns the updated reserves as well as the method ID of the opposite method as it is required when building the final searcher transaction.

The two Uniswap Methods differ in a few ways and thus have to be handled slightly differently. The first difference between the two being which field in the decoded transaction stores the amount of tokens being traded. In transactions swapping ETH for USDT, the $value$ field holds the amount of ETH being traded whereas the $amountIn$ field stores the amount of USDT being traded in transactions that swap USDT for ETH. As a result, when calculating the updated reserves, both functions have to be handled differently. Another way in which these functions differ is which token associated to the X and Y variable respectively. Since these variables represent the reserves of the token which the searcher trades in and receives, this differs based on the function of the target transaction since the searcher has to trade in the opposite token to the user in order to effectively backrun.

4.3.2 Calculating The Amount To Trade

Given the updated liquidity pools, the next step in calculating the profits is to utilise Equation 3.5 to calculate the amount of tokens the searcher must trade in to ensure a profitable backrunning operation. Due to Solidity's lack of support for floating point integers, the equation had to be slightly adjusted to accommodate this restriction.

Algorithm 8 attempts to calculate the amount a searcher should trade in by first calculating the numerator of equation 3.5. As part of calculating the numerator, the square root of the radicand must be calculated. Due to the lack of floating point integers in Solidity an encrypted version of the Newton method is implemented in the $sqrt$ function. This method utilises homomorphic addition to approximate the square root and is adapted from the openzeppelin sqrt function[2]. Following the calculation of the numerator, the calculation of denominator follows. In the fhEVM implementation of division ($TFHE.div$), the divisor must be plaintext. This, in combination with the fact that the divisor does not contain any sensitive, private details allows

[2] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/Math.sol

Algorithm 7 Token Reserve Calculation Algorithm

function UPDATELIQUIDITY(decodedTX, constants)
  if $methodID$ is $SwapExactTokensForETH$ **then** ▷ swapExactTokensForEth
    $searcherMethodID \leftarrow 0x7ff36ab5$ ▷ SwapExactEthForTokens method ID
    $uniswapFees \leftarrow TFHE.div(TFHE.mul(decodedTX.data.amountIn, 3), 1000)$
    $amountIn\_fees \leftarrow TFHE.sub(decodedTX.data.amountIn, uniswapFees)$
    $token1AfterSwap \leftarrow TFHE.add(constants.USDTInPool, amountIn\_fees)$
    $token2AfterSwap \leftarrow TFHE.mul(constants.EthInPool, constants.USDTInPool)$
    $X \leftarrow TFHE.div(token2AfterSwap, token1AfterSwap)$
    $Y \leftarrow TFHE.add(constants.USDTInPool, decodedTX.data.amountIn)$
  else ▷ SwapExactEthForTokens
    $searcherMethodID \leftarrow 0x18cbafe5$ ▷ swapExactTokensForEth method ID
    $uniswapFees \leftarrow TFHE.div(TFHE.mul(decodedTX.value, 3), 1000)$
    $amountIn\_fees \leftarrow TFHE.sub(decodedTX.value, uniswapFees)$
    $token1AfterSwap \leftarrow TFHE.add(constants.EthInPool, amountIn\_fees)$
    $token2AfterSwap \leftarrow TFHE.mul(constants.EthInPool, constants.USDTInPool)$
    $X \leftarrow TFHE.div(token2AfterSwap, token1AfterSwap)$
    $Y \leftarrow TFHE.add(constants.EthInPool, decodedTX.value)$
  end if
  return$(X, Y, searcherMethodID)$
end function

the protocol to leave the divisor unencrypted. Given both the numerator and denominator, the searcher amount in can then be calculated using the fhEVM division function.

Throughout this protocol, the fhEVM operators are utilised to perform calculations on the ciphertext handles. Before detailing the implementation of the of the operators, it is important to note that despite being represented as $uint256$ values, it is unlikely that an arbitrary $uint256$ value can be misinterpreted as a handle since handles are computed using cryptographically secure hash functions [6]. Similarly, this also means that handles of different ciphertexts are unlikely to have the same value as this would imply an insecurity in the $Keccak256$ hash function [6]. Operators in fhEVM utilise these handles in pre-compiled contracts to perform arithmetic, logic and comparison operations. To perform these calculations, the fhEVM functions call the corresponding operation in the TFHE-rs library [6].

4.3.3 Calculating Profits

Given the amount to trade in, the protocol moves to use equation 3.6 to calculate the profits from the backrunning trade. Algorithm 9 begins by calculating the amount received by the searcher after trading in the amount calculated in the previous section. The resultant amount, is substituted into equation 3.6 to calculate the searcher's profit. In the event where the profit calculation is less than the cost of arbitrage, the function returns a profit of 0 so as to not return a negative number. This function returns both the profit and the amount of tokens the searcher receives,

Algorithm 8 Amount To Trade Calculation Algorithm

function AMOUNTCALCULATION(const, X, Y)
$X \leftarrow TFHE.mul(X, prec)$
$priceFee \leftarrow TFHE.div(TFHE.mul(12, TFHE.mul(const.maxBuyPrice, Y)), 1000)$
$dividendRHS \leftarrow TFHE.sub(TFHE.mul(TFHE.mul(4, const.maxBuyPrice), Y), priceFee)$
$dividendSum \leftarrow TFHE.add(TFHE.div(TFHE.mul(9, X), 1000000), dividendRHS)$
$radicand \leftarrow TFHE.mul(X, dividendSum)$
$fee \leftarrow TFHE.div(TFHE.mul(X, 3), 1000)$
$amountDividend \leftarrow TFHE.sub(TFHE.add(sqrt(radicand), fee), TFHE.mul(2, X))$
$amountDivisor \leftarrow (2 * const.EthPrecision) - ((6 * const.EthPrecision)/1000)$
$amountIn \leftarrow TFHE.div(amountDividend, amountDivisor)$
return$(amountIn)$
end function

so that it can be utilised to ensure the searcher builds a valid transaction. Details of the validity checks follow in the proceeding section.

Algorithm 9 Profit Calculation Algorithm

function PROFITCALCULATION(decodedTX, X, Y, amountIn, const)
$amount\_fee \leftarrow TFHE.div(TFHE.mul(amountIn, 3), 1000)$
$x\_after\_fee \leftarrow TFHE.sub(TFHE.add(X, amountIn), amount\_fee)$
$y\_after \leftarrow TFHE.div(TFHE.mul(X, Y), x\_after\_fee)$
$amountOut \leftarrow TFHE.sub(Y, y\_after)$
$priceDiff \leftarrow TFHE.sub(const.minSellPrice, const.maxBuyPrice)$
$profit \leftarrow TFHE.div(TFHE.mul(priceDiff, amountOut), const.EthPrecision)$
if $profit \leq 2$ **then**
$profit \leftarrow TFHE.asEuint64(0)$
else
$profit \leftarrow TFHE.sub(profit, const.costOfArbitrage)$
end if
return$(profit)$
end function

Validity Criteria

As described in the previous chapter, guaranteeing the validity of both the user transaction and the calculated amount is essential to ensure that the block builder receives a profitable and valid transaction which is capable of successfully extracting value from the user's transaction.

Algorithm 10 begins by maintaining a validity boolean ($valid$) which remains True so long as the user transaction, calculated amount, and profit parameters meet the necessary criteria. The user transaction's criteria contains a series of differing comparisons, which are performed before the amount calculation, to ensure that the user has passed in a valid Uniswap transaction with the correct destination address,

deadline and path length. After the calculation of the profit, the searcher's computation criteria tests whether the calculated amount to be traded is greater than 0 and less than the respective reserves, since it is not possible to trade more tokens than is available in the reserve at any given time. The searcher's criteria further ensures that the transaction is profitable so as not to facilitate non-profitable transactions being deployed. In the case of an invalid transaction, the protocol returns an empty transaction to the user.

Algorithm 11 deciphers and creates the transaction that is to be RLP encoded and sent to the searcher. The first case that algorithm 11 addresses is the case of a valid transaction and computation. In this case, the algorithm builds the transaction for both methods accordingly. If the searcher invokes the swapExactTokensForEth method, the algorithm sets value field to zero since no ETH is being sent. Since ETH is sent in the case of the searcher invoking the swapExactEthforTokens method, the amount in field is in turn set to zero and the searcher's calculated amount is passed to Uniswap through the value field. In both cases, the user's token fields are swapped to ensure that the searcher is able to effectively backrun the transaction. Finally, in the case of an invalid transaction, the algorithm creates an empty transaction that gets encoded and sent back to the searcher.

Algorithm 10 Validity Comparison Algorithm

if $decodedTX.to = UniswapRouter \ \& \ decodedTX.data.deadline \geq minDeadline \ \& \ decodedTX.data.pathLength \geq 2$ **then** ▷ User Transaction Criteria
    $valid \leftarrow True$
else
    $valid \leftarrow False$
end if
    [...]
if $profit > 0 \ \& \ 0 < amountIn < reserves$ **then** ▷ Searcher Transaction Criteria
    $valid \leftarrow True$
else
    $valid \leftarrow False$
end if

4.4 Creating The Backrunning Transaction

Once the amount of tokens that the searcher should swap is calculated, the next and final step of the protocol is to build the serialised backrunning transaction. The transaction that is returned to the searcher is dependent on which Uniswap method is called. If the searcher calls the "swapExactTokensForEth" method: the value field of the transaction is set to zero, the first token field of the transaction is set to the address of the non-Ethereum token while the address of the second token is set to the address of the Ethereum token. As for the "swapExactEthForTokens" method: the amount traded in field is now set to zero, the first token address is set to the

Algorithm 11 Transaction Building Algorithm

if $valid$ is $True$ **then**
  if $methodID$ is $SwapExactTokensForEth$ **then**
    $data \leftarrow (methodID, amountIn, amountOut, 160, address, deadline, 2, token2, token1)$
    $tx \leftarrow (nonce, gasPrice, gasLimit, to, 0, data)$
  else
    $data \leftarrow (methodID, 0, amountOut, 128, address, deadline, 2, token2, token1)$
    $tx \leftarrow (nonce, gasPrice, gasLimit, to, amountIn, data)$
  end if
else ▷ Invalid Transaction
  $data \leftarrow (0, 0, 0, 0, 0x0, 0, 0, 0x0, 0x0)$
  $tx \leftarrow (0, 0, 0, 0x0, 0, data)$
end if
return$(encodeTX(tx))$

address of the Ethereum token and the second token address is now set to that of the non-Ethereum token.

Algorithm 12 implements the designed transaction encoding scheme as described in the previous chapter. This algorithm uses the solidity-rlp-encode library [35] to first encode the nonce, gas price, gas limit and the destination address fields. These fields are then appended to the $txArray$ array. Following this, Algorithm 12 builds the data field by first appending the $methodID$ in addition to the ABI encoding of the $amountIn$ field to the $dataArray$ variable, in the case that the function being called is the swapExactTokensForEth method. Then, the ABI encodings of the amount out, address offset, destination address, deadline, address length and both token addresses are all concatenated and RLP encoded using the $encodeBytes$ function. This encoding of the $dataArray$ variable is then appended to the $txArray$ array alongside the previous fields. Finally, $txArray$ is further encoded and returned to the user as a raw, serialised Ethereum transaction that exploits price shifts of the user transaction to extract value and re-balance the network.

In the fhEVM framework the decryption operation requires differing behaviours from validators and full nodes. While validators can engage in a threshold decryption protocol with other validators to decrypt the ciphertext, full nodes are unable to do this since they do not possess a share of the secret key [6]. To work around this, validators store the resulting plaintext value on chain after every decryption where full nodes can use these values to decrypt the ciphertext [6]. Given this, it is important to note that prior to the backrunning transaction being encoded, the encrypted variables are implicitly decrypted in order to create a valid Uniswap transaction since it cannot decipher transactions with encrypted fields. That being said, the fields of the transaction are easily left encrypted by simply omitting the $TFHE.decrypt$ function call. This would ensure that the private states calculated in the protocol are left encrypted in the final transaction and can be part of an encrypted five transaction bundle containing the three non-target user transactions, followed by the user's target transaction and finally ending with the calculated backrunning transaction. This

removal of the decryption would also decrease the gas consumption of the transaction by around 1,000,000 units as well as optimise the runtime of the transaction.

The need for optimisation, it's techniques and their implementation into this protocol is analysed in detail in the following chapter.

Algorithm 12 Transaction Encoding Algorithm

function ENCODETX(transaction)
$txArray[0] \leftarrow transaction.nonce.encodeUint()$
$txArray[1] \leftarrow transaction.gasPrice.encodeUint()$
$txArray[2] \leftarrow transaction.gasLimit.encodeUint()$
$txArray[3] \leftarrow transaction.to.encodeUint()$
$dataArray[0] \leftarrow transaction.data.methodID$
$i \leftarrow 1$
if $methodID$ is $SwapExactTokensForETH$ **then**
$txArray[4] \leftarrow uint(0).encodeUint()$
$dataArray[1] \leftarrow abi.encode(transaction.data.amountIn)$
$i \leftarrow 2$
else
$txArray[4] \leftarrow transaction.value.encodeUint()$
end if
$dataArray[i] \leftarrow abi.encode(uint(transaction.data.amountOutMin))$
$dataArray[i+1] \leftarrow abi.encode(transaction.data.addressOffset)$
$dataArray[i+2] \leftarrow abi.encode(transaction.data.dataTo)$
$dataArray[i+3] \leftarrow abi.encode(transaction.data.deadline)$;
$dataArray[i+4] \leftarrow abi.encode(transaction.data.addressLength)$
$dataArray[i+5] \leftarrow abi.encode(transaction.data.token1)$
$dataArray[i+6] \leftarrow abi.encode(transaction.data.token2)$
$dataConcat \leftarrow bytes.concat(dataArray)$
$txArray[5] \leftarrow dataConcat.encodeBytes()$
return$(txArray.encodeList())$
end function

Chapter 5

Optimisation

Given the successful design and implementation of the private backrunning protocol it is essential to optimise this solution to ensure an efficient and workable protocol. This chapter explains the need for, techniques of, and implementations of optimisation in the context of the backrunning solution. Section 5.1 introduces the chapter by describing the need for optimisation generally and specifically in the context of Ethereum and smart contracts. Following this, section 5.2 describes the available optimisation techniques as well as their efficacy in this context. Section 5.3 then outlines the implementation of these techniques within the backrunning solution, further describing their effects on the project.

5.1 The Need For Optimisation

In Ethereum, applications known as smart contracts are executed on the blockchain, where the price for executing such applications is measured using gas. Gas allocates portions of the Ethereum virtual machine (EVM) to allow for the executions of transactions [38]. More complex transactions require commensurately higher allocations of the EVM to run, and thus require a higher gas amount. As a result, it is important when developing any application intended to be run on the blockchain that the developer aims to reduce the amount of gas consumed to minimise any unnecessary expenditure as well as to mitigate the possibility of malicious attacks [38].

With regard to the backrunning protocol, optimising the solution is of paramount importance due to the time critical nature of the operation as well as given that the gas cost incurred reduces the searcher's final profit. As a result, optimising the solution ensures that the searcher receives the maximum profit, further incentivising the use of this protocol. In addition to economic factors, a lack of optimisation leads a greater computational overhead and slower transaction times which may effect the profitability and eligibility of the backrunning extraction due to the dynamic nature of the token reserves. Therefore ensuring optimal transaction times leads to more effective backrunning and profitability for the searcher.

5.2 Optimisation Techniques

In order to effectively optimise the backrunning protocol is it important to outline the different gas-efficient methods available for use. This section explains these methods in three categories: data types and operations in section 5.2.1, storage in section 5.2.2 and loops in section 5.2.3.

5.2.1 Data Types and Operations

All states are declared with a specific data type in Solidity, which cannot be changed once it has been set since Solidity is a statically typed language, which also means that certain data types such bytes or fixed size variables are more gas efficient that other data types like strings and dynamic sized variables [38]. As a result, the choice of which data type to use to represent a variable can significantly affect the gas consumption of a protocol.

An example of how to reduce gas consumption with regard to data types in smart contracts is the use of the $uint256$ data type over other unsigned integers of lower precision. When a variable is declared as an unsigned integer of lower precision, it is first converted to the $uint256$ format before being converted back, thus consuming more gas due to the extra conversion. To mitigate this, using the $uint256$ data type instead of other smaller data types will reduce the conversion and save gas as a result [38]. Similarly, the use of the $bytes$ data type over the $string$ data type can further lower the gas consumption since strings consume more gas and must fit inside 32 bytes in order to prevent wasting memory. Generally, in Solidity, variables with a fixed size are more efficient and consume less gas compared to variable sized variables. Fixed size variables are pre-allocated a fixed amount of memory whereas variables of flexible size are dynamically allocated memory thus consuming more gas. Finally, the last optimisation technique with regard to data types is the explicit initialisation of variable with a value of zero. In Solidity, when a variable is not assigned a value, it is implicitly assigned a default value of zero. Explicitly assigning this for a variable is both unnecessary and inefficient due to the increased amount of gas consumed [38].

Mathematical operations in Solidity each have a respective cost in terms of gas consumption, which can be optimised for efficiency in a few ways. One technique is by limiting the number of operations performed for an expression through its factorisation and re-arrangement. Secondly, since not every operation is equal in gas consumption it is essential to rearrange expressions to reduce the amount of expensive operations through the grouping of lower complexity operations. This method is especially crucial in this context since encrypted operations consume a greater amount of gas compared to regular operations in Solidity. The gas fees for the fhEVM operations using $euint64$ (which can be seen in Figure 5.1), compared with regular EVM operations (seen in Figure 5.2), are significantly more expensive, costing 128,200 times more to do an encrypted multiplication than it is to perform its unencrypted counterpart. Finally the last technique to lower gas consumption is to hardcode a variable with the the result of a known operation instead of calculating

euint64

Function name	Gas fee
`add` / `sub`	188,000
`add` / `sub` (scalar)	188,000
`mul`	641,000
`mul` (scalar)	356,000
`div` (scalar)	584,000

Figure 5.1: fhEVM euint64 gas fees [3]

Stack	Name	Gas	Initial Stack
00	STOP	0	
01	ADD	3	a, b
02	MUL	5	a, b
03	SUB	3	a, b
04	DIV	5	a, b

Figure 5.2: EVM gas fees [5]

the same result at every run. Despite resulting in less understandable code, this technique greatly improves gas consumption.

5.2.2 Storage

The use of storage for storing data in smart contracts can greatly affect the gas consumption, therefore optimising this area of the smart contract can yield great improvements with regard to efficiency. One area of optimisation is in the writing and reading to slots in memory. In the EVM, memory is organised in slots, each of which being at least 32 bytes [38]. This means that slots still consume the same amount of gas to access, regardless of whether they are completely or partially filled. As a result, minimising the number of slots accessed by ensuring that slots and completely filled can greatly improve gas-consumption. This concept of minimising the slots is known as packing and it applies to both structs and variables [38].

Struct packing in smart contracts applies to structs, which serve as a data type

that is used to group other related data types. Within structs each individual data type is stored in its own memory slot. Due to this, as the number of individual data types increases so does the gas consumption of the struct. Therefore in order to optimise the gas consumption when using structs, rearranging and packing the individual data types can significantly reduce the number of memory slots required and therefore also the resulting gas consumption [38]. Similarly, variable packing also aims to optimise the storage and access of state variables in smart contracts by rearranging their order to reduce the number of memory slots used.

Struct and variable packing work similarly with the fhEVM as it does with the regular EVM as stated in the fhEVM whitepaper, "When encrypting a plaintext of many bits, it is possible to use a packing technique to compress the bundle of ciphertexts resulting from such an encryption." [6].

5.2.3 Loops

In Solidity, loops and can have significant impacts on gas consumption. As a result, to limit the impact of loops one should minimise the use of loops and especially the use of nested loops as they consume significantly more gas than singular loops. Furthermore, in the case that the use of loops is inevitable, one should simplify their use by performing some of the earlier mentioned techniques on data operations and data types, in order to minimise repetitive operations within loops. As a result, this can significantly reduce gas consumption in smart contracts. Expensive loops that require large amount of calculations should rather be performed off chain by other languages that are better equipped at handling them such as Python or Java Script [38].

5.3 Implementation of Techniques

Given the existence of the optimisation techniques described above, it is important to implement them into the backrunning algorithm, in order to maximise program efficiency and minimise overall gas consumption to a realistic, appropriate level. This section of the study outlines the implementation of different optimisation techniques and their effects on the total gas consumption.

The first implementation of the data types optimisation techniques in the backrunning protocol is the use the fixed-size variables over variable sized variables. Throughout this project all non-encrypted integers are represented using the $uint256$ data type, in order to avoid the unnecessary conversion to $uint256$ done by other, smaller unsigned integers. Similarly, to minimise the number of memory slots the project utilises, the $bytes$ data type is used to represent strings rather than using the $string$ data type as a result of its fixed nature. Implementing these techniques into the project in addition to the use of fixed size arrays over dynamic ones reduced the gas consumption by around 40,000 units of gas. Furthermore, the use of optimisation on data operations greatly reduced the total gas consumption of the project. The first optimisation technique implemented was the reducing of gas consumed by equations 3.5 and 3.6 by factorising and rearranging the equation to limit the

number of expensive operations performed. Additionally, in order to further reduce the gas consumption the result of some calculations were hardcoded into the equation, further reducing the gas consumption. An example of this can be see when calculating the radicand in equation 3.5, where the result of the factorisation and re-arrangement can be seen in equation 5.1 which greatly decreases the number of expensive operations. Furthermore, another example of this is seen in the denominator of equation 3.5 where the Uniswap fees were hard coded into the equation rather than performing a separate calculation. This optimisation of data operations resulted in a gas reduction of around 1,000,000 units of gas.

$$radicand = \frac{9 \cdot x + 3988000 \cdot price \cdot y}{1000000} \tag{5.1}$$

Struct packing and variable packing were also utilised to optimise this project by re-arranging the order of data types to ensure optimal allocation of memory slots. An example of this can be seen in the $decodedTransaction$ struct where all of the unsigned integers are grouped together followed by the encrypted integers and bytes. Similarly, the $searcherConstants$ struct also implemented this struct packing by grouping all of the unsigned integers to ensure that all of the assigned memory slots are full, and none are left partially filled. Struct packing and variable packing reduced the gas consumption of the code by 20,000 units of gas, a sizeable amount.

Finally, the last optimisation techniques utilised during development were the reduction and minimisation of loops, firstly by omitting any unbounded loops to prevent the potential for an unpredictable amount of gas expenditure. Furthermore, throughout the project all unnecessary loops were omitted apart from three. The first of these exemptions being a necessary helper function which extracts bytes from one larger variable to another smaller one and the other two being in a nested for loop which is responsible for combining multiple transactions to find the optimal transaction ordering. Despite generally being considered antithetical, due to their high gas usage, this use case necessitated their use due the need to iterate over all of the transactions. Similar to struct and variable packing, loop optimisation reduced the gas consumption by 20,000 units of gas.

Optimising the code base using the above techniques in addition to the general code optimisation resulted in a reduction of around 1.1 million units of gas (see Table 5.1). As previously discussed, this reduction greatly aids the final user to exercise their backrunning strategies by providing an efficient and usable final product.

Summary of Gas Saving by Optimisation Techniques	
Optimisation Technique	**Gas Saved**
Data Types	40,000
Data Operations	1,000,000
Packing	20,000
Loops	20,000
Total Gas Saved	1,080,000

Table 5.1: Summary of Optimisation Techniques

Chapter 6

Experiments and Results

The preceding chapter built upon the initial implementation of the solution by optimising the project with the purpose of reducing the gas consumption and improving the general efficiency. This chapter now moves to detail the experiments done on the system and the metrics used to measure the systems effectiveness and efficiency.

This chapter begins by outlining the experimental setup in section 6.1, where the test cases, metrics and experiments are discussed. Subsequently, section 6.2 details the results of the described experiments before concluding the chapter with an end-to-end proof of concept in section 6.3.

6.1 Experimental Setup

In order to test the validity and effectiveness of the final solution, it is important to carefully design a set of experiments that analyses and tests all cases of the solution. This section, describes the test cases utilised, the metrics and methods of measurement.

To effectively test the contributions and the achievements of the solution, three test cases were run. The first test case contains four transactions trading ETH to USDT. The second test case contains four transactions trading USDT for ETH and the final test case contains an even combination of the previous two test cases. These test cases were designed to address all edge cases of the solution. In addition to the four transactions, all test cases will be executed with equal values of maximum buy price and minimum sell price. Similarly, all of the test cases are run with the initial ETH reserves at 22.7 thousand and the initial USDT reserves at 55.9 million, which represent the live reserves as of August 2024 [4]. These strategic constants remained unchanged throughout the test cases to create a controlled experiment where any discrepancies in the findings are solely based on the transactions and no other external parameters. It is important to note that the searchers' strategy is already pre-set to a decimal precision of 6.

The first case will consist of the four transactions trading in 3, 4.5, 6.25 and 1.125 ETH for USDT. The second test case will, again, consist of four transaction, these however trading in USDT for ETH with each transaction trading in 5321.74, 4831.02, 10831.67 and 2109.10 USDT respectively. The final test case will similarly

consist of 4 transactions, The first two trading 3 and 4.5 ETH for USDT and with the last two trading in 5321.74 and 4831.02 USDT for ETH.

In order to effectively measure effectiveness, the tests' metrics to be measured included each transaction's gas consumption and runtime. These test cases were run using the Remix IDE with the contracts being deployed on the ZAMA Devnet.

6.2 Summary of Results

Tables 6.1, 6.2 and 6.3 showcase the results of the three, previously discussed test cases, while Figures 6.1 and 6.2 summarise the results of the gas consumption and runtime respectively. The gas consumption across the three test cases remained similar with only a small dip in Test Case 2 when performing the $updatePools$ function. This dip can be attributed to a lower amount of operations since when updating the pools for ETH to USDT trades, the ETH value is made to match the precision of USDT. This division operation is not necessary when trading USDT for ETH and as a result, explains the slight dip in gas consumption especially since encrypted division consumes significantly more gas than other encrypted operations.

As for the runtime comparison, this was again relatively stable across Test Cases 1 and 2 while Test Case 3 took around twice as long to combine the transactions. This is attributable to the mixing of different transaction types resulting in longer runtime. Additionally, Test Case 1 resulted in a runtime twice as long as the other test cases when building the final user transaction, which is due to the type of transaction being built. This is since the transaction being created for Test Case 1 differs from the other two cases, since it trades USDT for ETH whereas the others trade ETH for USDT in the backrunning transaction. This justifies the increase in runtime due to the additional computations necessary to decrypt and encode the amount in field present when trading USDT for ETH but not in the reverse transaction.

Unsurprisingly, as the most complex method of the entire protocol, the combineTransactions method consumed the most gas unanimously across all three test cases. Averaging around 9.8 million gas across the test cases, this method is close to reaching the 10 million gas limit present in the ZAMA devnet where this experiment was executed. It was clear from the initial design process and the implementation that this method was bound to be the computational bottleneck, due to the nested loop where in which the addition was performed. However, the short relative run time of the method, was surprising. Despite consuming significantly more gas than any of the other methods, the combineTransactions method had the shortest runtime of any of the methods across all of the test cases.

Moreover, the updatePools and amountInCalc functions both had similar gas consumption and runtimes across the test cases, which is unsurprising due to the amount and nature of the operations within both of these methods. The updatePools and amountInCalc functions exhibited similar numbers of encrypted operations, with commensurately negligible difference in gas consumption. The slight difference in runtime between them in Test Cases 1 and 3 could be attributed to variety of reasons, including network congestion, which often slightly affects the runtime of transactions. Finally, when analysing the results of the profitCalc functions, this had sig-

nificantly lower gas consumption than the other methods as a result of the decreased amount of encrypted operations compared to amountInCalc and updatePools. A surprising result, however, is that despite the reduced computations, the runtime of this method was still on par with the other functions and occasionally exceeded them in the case of the buildSearcherTX function in Test Case 1. This increased runtime could again be a result of external factors. With regard to the buildSearcherTX function, this consumed around 1.5 million gas despite not containing any encrypted operations. This is ultimately due to the implicit decryption needed to build the transaction through the use of the $TFHE.decrypt$ function which alone consumes 5000,000 gas [3]. Having been called three times to decrypt the amount in, value and amount out states, this details the source of the increased gas consumption and runtime of the transaction. Given these results, it is important to note that since the fhEVM execution flow might depend on the decryption of encrypted values which cannot happen during local gas estimation due the need for validator interaction, local gas estimation is less precise when encrypted data is used compared to when only plaintext data is being utilised [6].

Overall, the runtime of the transactions are notably fast compared to other FHE implementations but when comparing with the fhEVM benchmarks (seen in Figure 6.3), the runtime of the transactions are consistent with these benchmarks considering the cumulative effect of the operations in the transactions. These faster runtimes are as a result of the previously mentioned optimisations which resulted in noticeable improvement due to the minimisation of expensive encrypted operations. An anomaly in the outlined runtimes however, is the execution of the buildSearcherTX in the first test case which demonstrated a longer runtime despite the lack of encrypted operations and as mentioned above this anomaly could be as a result of the other external factors or simply due to the imprecise fhEVM estimations resulting from the decryption of the private fields.

Test Case 1		
Method Name	**Gas Consumption**	**Runtime (seconds)**
combineTransactions	9784909	3.91
updatePools	7825456	6.82
amountInCalc	7989849	9.84
profitCalc	4284904	8.71
buildSearcherTX	1520893	14.40

Table 6.1: First Test Case

6.3 Proof of Concept

While it is important to test the efficiency of the protocol, it is equally as important to test the effectiveness. This section of the study moves to perform this by showcasing an end-to-end proof of concept where multiple user transactions are inputted to the protocol in addition to the searcher's strategy to produce a profitable transaction capable of extracting additional value from the target transaction.

Test Case 2		
Method Name	**Gas Consumption**	**Runtime (seconds)**
combineTransactions	9820905	3.82
updatePools	7251953	9.89
amountInCalc	7989849	9.96
profitCalc	4178438	9.79
buildSearcherTX	1589420	7.82

Table 6.2: Second Test Case

Test Case 3		
Method Name	**Gas Consumption**	**Runtime (seconds)**
combineTransactions	9781236	7.12
updatePools	7825456	9.63
amountInCalc	7989849	7.71
profitCalc	4389638	8.78
buildSearcherTX	1514920	7.71

Table 6.3: Third Test Case

This proof of concept is going to be utilising the first test case outlined in section 6.1 of the report. An execution of the protocol consists of five functions: combineTransactions, updatePools, amountInCalc, profitCalc and buildSearcherTX.

combineTransactions

The first step of our protocol takes the four user transactions and returns the most profitable target transaction. This is done by evaluating each transaction's relative trade size and using that value as an indicator for profitability. This function returns the target transaction as well as the updated reserves to which the remaining three transactions were applied.

Given the inputted transactions from the first test case and the searcher's parameters, the combineTransactions function returns the following:

- **The target transaction's value**: 6.25 ETH
- **The updated ETH reserves**: 22708625000 (22708.625000)
- **The updated USDT reserves**: 59877249283036 (59877249.283036)

updatePools

Given the target transaction and the updated reserves, the next step in the protocol is to update the reserves by applying the newly identified target transaction on the calculated reserves to receive an updated total of the amount of ETH and USDT in the pool. This step in the protocol is necessary to calculate the searcher trade size and profit as is shown in the following sections.

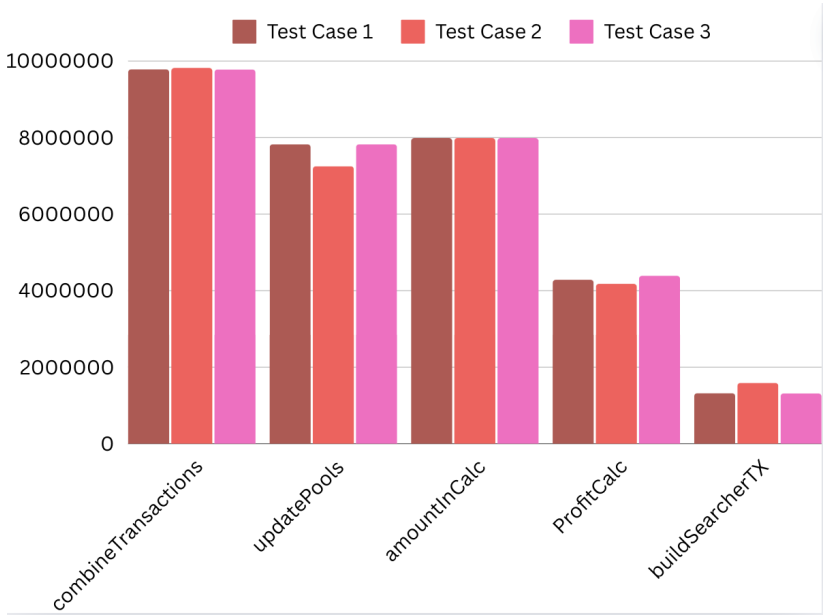

Figure 6.1: Test case gas comparison

This method returns the X and Y variables which are used in equation 3.5 to calculate the amount of tokens the searcher should trade. These variables represent the token reserves in the pool where X is the token to sell and Y is the token to buy. In this instance X represent the updated USDT reserves while Y represents the updated ETH reserves.

Given this, the function returns the following:

- **The updated USDT reserves (X)**: 59860483653236 (59860483.653236)
- **The updated ETH reserves (Y)**: 22714875000 (22714.875000)

amountInCalc

Given these updated reserves, the next step is to use equation 3.5 to calculate the optimal amount of USDT the searcher should sell.

The result of this calculation is the following:

- **The amount to trade in (USDT)**: 4724073203465 (4724073.203465)

profitCalc

The result of the previous section combined with the maximum buy price, minimum sell price and the pool reserves are inputted into equation 3.6 to calculate the searcher's potential profit.

The result of this calculation is the following:

- **The amount received (ETH)**: 22714659763 (22714.659763)
- **The profit (USDT)**: 4267915689290 (4267915.689290)

Figure 6.2: Test case runtime comparison (seconds)

buildSearcherTX

In the case of a profitable backrun, this section encodes the backrunning transaction to be returned to the searcher. In the case of a unprofitable transaction, an empty transaction is then returned to the searcher.

In this test case, the searcher is able to extract a profit of 4267915.69 USDT when trading in 4724073.20 USDT for 22714.66 ETH. This is assuming that the searcher is able to sell ETH for at least 3500 USDT/ETH at a different exchange that carries a cost of 2 USDT.

To confirm these transaction on chain the searcher must create a bundle of 5 transactions, consisting firstly of the 3 non-target transaction, followed by the user's target transaction and the searcher's backrunning transaction.

Operation	Platform	`uint8`	`uint16`	`uint32`
Addition	CPU	78 ms	113 ms	141 ms
Subtraction	CPU	81 ms	111 ms	139 ms
Multiplication	CPU	127 ms	227 ms	368 ms
Division[5]	CPU	226 ms	322 ms	544 ms
Equality	CPU	40 ms	40 ms	65 ms
Comparison	CPU	58 ms	80 ms	110 ms
Bitshift	CPU	107 ms	138 ms	200 ms

Figure 6.3: "Benchmarks of typical FHE operation on encryption of unsigned integers of various sizes. CPU times are multi-threaded using an Intel Xeon Gen 3 processor on AWS m6i.metal" [6]

Chapter 7

Evaluation

This chapter moves to analyse the performance of our solution by first discussing the assumptions taken in this project in section 7.1 and the apparent limitations of this project in section 7.2. Following this, section 7.3 compares the approach taken in this study to that of previous solutions by Flashbots.

7.1 Assumptions

The results of this solution are naturally only valid under a few assumptions. The first of which being that arbitrage opportunities do exist in the UniswapV2 decentralised exchange. This assumption is necessary to make since the lack of any arbitrage opportunities would make backrunning transactions pointless and thus, this assumption forms the foundation upon which this thesis is based.

The remaining assumptions that must be made about this project are technical assumption that relate to the searcher and their practical use of this solution. The first of these assumes that the searcher is able to obtain the user transaction and has full access to them. This assumption is necessary since it allows the searcher to utilise and simulate the pending transactions for profitable gain. This assumption does not, however, mitigate the ability of other searcher's to have the same ability to simulate the user's transaction and build their own plaintext bundle. Similarly, this thesis also makes the assumption that the searcher is able to retrieve accurate and up to date pool information including both the reserves and the price of the tokens. This assumption mitigates the need for the use of an oracle to retrieve the current pool information, which would significantly increase the complexity and gas consumption of the solution.

7.2 Limitations

In addition to the assumptions made by this project, there also some limitations of this final solution. The first limitation of this solution is the fact that it can handle integers only up to 64 bits of precision. This limitation comes as a result of the fhEVM framework not supporting larger encrypted integers (as of fhEVM version

0.4). Consequently, the decimal precision for ETH, had to be reduced to a decimal precision of 6. Similarly, this restriction on the integer size forced a change in the calculation of the searcher's trade size and profit in order to prevent an overflow. This ultimately increased the overall number encrypted operations, only adding to the gas consumption.

Another limitation of this project is that due to the deployment being on the ZAMA devnet. Contracts had a gas limit of 10 million meaning that the protocol could not be presented as one end-to-end transaction which takes in the user transactions and the searcher's strategy and outputs the backrunning transaction. Rather, it necessitated the need for the protocol to be separated across five transactions which, significantly reduced the usability for the searcher, only increasing the difficulty encountered to successfully and efficiently backrun transactions. However with the inevitable introduction of the fhEVM Co-Processor, developers will have the ability to write confidential applications on any non-FHE enabled chains [3] and thus will not be limited by the devnet gas limit. The future developments of this thesis is discussed in more detail in Section 8.3.

Due to the desired minimisation of gas consumption, the number of transactions that can be combined was limited to four, in the interest of finding a balance between efficiency and usability. While this increases the complexity of searcher's strategy compared to previous solutions, an ideal solution would offer the combination of more transactions. The minimisation of gas consumption also entailed limitations in terms of the choice of which parameters to encrypt and which to keep as plaintext. Ideally, all parameters associated with the user's transactions and the searcher's strategy would be encrypted however due to the gas limitations, only the private, sensitive parameters were encrypted while the rest remained as plaintext.

7.3 Comparison With Previous Solutions

When comparing this study's solution with the previous two Flashbots solutions, there are naturally numerous similarities. The first of which being the equation used to ascertain the searchers trade size. This equation was originally used in the Flashbots MPC solution [30] to extract the optimal amount to trade in. In addition to this, this study's solution offers a similar structure to that of the MPC solution [30] whereby the user's transactions were first decoded then the searcher's strategy was loaded followed by the execution of the computations and comparisons. This thesis further followed from the foundations set by Flashbots by using the UniswapV2 decentralised exchange as the target of the backrunning.

In addition to this, this project advances the work done by Flashbots in a plethora of ways, including the acceptance of multiple user transactions. Both the Flashbots solutions discussed in sections 2.3 could only accept one target transaction which only limited the searcher's ability to create complex transactions. Thus, this project built upon Flashbots' recommendation for future work to handle multiple input transactions since searchers "often combine more than one transaction to generate novel MEV opportunities" [8]. Another aspect in which this project extends the original MPC solution is by allowing searchers to backrun more one than Uniswap

method. The original MPC solution only allows searchers to backrun user transaction which call the “swapExactEthForTokens” method whereas this solution also allows searchers to backrun user transaction which call the “swapExactTokensForEth” method, and thus advancing previous solutions through the expansion of the protocol’s usability.

Additionally, a major difference from previous solutions is that this solution, as described in section 3.1, deploys the backrunning protocol on the public network with the advantage of the computations being run on the network validator’s machine rather than on the searcher’s machine. This differs from the previous solutions where the backrunning protocols were deployed locally on the searcher’s machine. This solution has the added benefit of increasing the transparency for the user as well as making use of high performing, secure machines. This difference enables new strategies by reducing the cost necessary to perform the arbitrage. This is since the computational burden has now been offloaded to the validator, thus lowering the computational bar necessary to become a searcher due to the lack of need for sophisticated hardware.

Finally, the last major difference between this solution and the previous solutions is that this solution built upon the future work outlined in the MPC solution by allowing for a high-level searcher language which was implemented using both Solidity and the fhEVM framework. The MPC solution only offered a low level implementation which was “prone to errors” [30] and as a result, this project built upon this by using the high-level fhEVM framework to implement the backrunning solution.

Chapter 8

Conclusion

The final chapter of this thesis summarises the carried study by first outlining the achievements of this research in section 8.1, then by discussing the ethical, legal and professional considerations in section 8.2. Following this, this final chapter moves to discuss the potential future developments of the backrunning solution in section 8.3 before summarising the thesis and the chapter in section 8.4.

8.1 Summary of Achievements

This project built upon previous solutions by utilising the fhEVM framework to allow searcher's to blindly backrun user transactions. In addition to this, this project also successfully achieved the following:

- Allowed the searcher to input and combine multiple user transactions.
- Deployed the solution on the public network.
- Accepts user transactions calling both the "swapExactEthForTokens" and "swapExactTokensForEth" methods.

8.2 Ethical Considerations

This study's main aim was to create a tool to allow searchers to blindly backrun user transactions. This solution was developed with the intention of ethical use to allow users to benefit from their transactions being leveraged, searchers to benefit for creating the backrun transaction and markets and the larger Ethereum environment to benefit due to the resulting stabilisation. Despite this, there are a few ethical considerations associated with the outcomes that need to be outlined. By outlining these ethical issues and ensuring that my solution does not introduce new vulnerabilities, this carried study can positively contribute to the blockchain community by creating a fairer, more just ecosystem where the rights of all users are protected. All full checklist of the Ethical considerations can be seen in Appendix B.

The first ethical consideration is the processing of personal / sensitive data. Hence, within this study the fhEVM framework is used which provides "unprecedented levels of privacy and security" [3]. ZAMA provides this through the use of FHE to compute private states directly on chain, MPC for the threshold decryption of FHE ciphertexts and ZK-proofs to ensure the integrity of encryption and decryption [3]. The use of this framework will then ensure ethical usage and processing of personal data.

The second ethical consideration is the resulting effect on the user and the markets. As a result, this study focuses on backrunning and arbitrage instead of frontrunning or sandwich attacks to ensure that this solution cannot be used in a way that is harmful to users. This approach also ensures that the user experiences lower transaction fees and improved delays. Furthermore, focusing on arbitrage results in positive, stabilising effects for the markets thus making sure that this solution does not inadvertently create newer forms of market manipulation, MEV or other unfair disadvantages for users.

8.3 Future Work

Whilst successfully achieving the outlined aims and objectives, this project still leaves space for some possible future developments, as suggested below:

1. **fhEVM Co-Processor**
   During the development of this project, the fhEVM Co-processor by ZAMA was announced. This would allow developers to write confidential application on any non-FHE enabled chain [3]. This Co-processor further allows for up to 256 bit integers and a full range of operators which would mitigate some of the limitations faced during the development of this project. Furthermore, the Co-processor also removes the need for bridging between the Co-processor and the blockchain since all assets are onchain on the L1 [3]. In addition to this, the Co-processor also allows for the mix of plaintexts and ciphertexts in operations, allowing for greater scalability [3].

2. **Different Chains**
   In order to efficiently allow searchers to apply novel search strategies, a future development of this project would be to extend its applicability further from Uniswap and Ethereum and to allow for cross chain arbitrage.

3. **Encrypted block building**
   As a natural development, utilising FHE to allow builders to perform encrypted block building will ensure that two agents in the transaction supply chain are able to act privately, thus enhancing the onchain privacy and security.

4. **Efficiency**
   Finally, an important future development of this project is to improve upon the efficiency and usability of this project for the searcher by minimising both the gas consumption and the runtime of the protocol by utilising the ever improving technologies such as fhEVM Co-processor and the fhEVM version 0.5.

8.4 Summary

In summary, this project has achieved its intended aims and objectives in using FHE to allow searchers to blindly backrun private user transactions.

This project successfully utilised Solidity and the fhEVM framework to design, implement and optimise a backrunning solution which allows searcher's to apply their strategies on pending transaction to extract additional value. This project expanded on previous solutions by accepting multiple user transactions to create novel search strategies. This program also successfully handles transaction calling both the "swapExactEthForTokens" and the "swapExactTokensForEth" methods all whilst being deployed on the public network.

The conducted research addresses the shortcomings of existing solutions by leveraging FHE to create a privacy preserving mechanism allowing searchers to extract value from user transactions without compromising their own or the user's privacy. Offering a more secure alternative to previous solutions, this research demonstrates the practical usability of FHE in decentralised finance.

Bibliography

[1] etherscan.io. Ethereum daily transactions chart — etherscan, 2024. pages i, 26, 27

[2] Flashbots . Flashbots transparency dashboard, 2024. pages i

[3] Zama . Welcome to fhevm — 0.4 — fhevm, 05 2024. pages i, v, 22, 23, 51, 56, 61, 64

[4] Uniswap interface. pages v, 10, 54

[5] Opcodes for the evm, 2024. pages v, 51

[6] ZAMA, Morten Dahl, Clément Danjou, Daniel Demmler, Tore Frederiksen, Petar Ivanov, Marc Joye, Dragos Rotaru, Nigel Smart, and Louis Tremblay Thibault. fhevm/fhevm-whitepaper.pdf at main · zama-ai/fhevm, 2022. pages v, 40, 44, 47, 52, 56, 59

[7] Monoceros Ventures . The mev book, 2023. pages 1, 2

[8] Flashbots . Leveraging homomorphic encryption for maximally extractable value (mev) mitigation: Enabling blind arbitrage on decentralised exchanges, 2024. pages 3, 19, 20, 61

[9] EigenPhi. Thriving amidst turmoil: the annual key takeaways for mev performance in 2022, 2023. pages 3

[10] Yehuda Lindell. Secure multiparty computation. *Communications of the ACM*, 64, 2021. pages 3

[11] Stephane Gosselin and Ankit Chiplunkar. The orderflow auction design space, 2023. pages 3

[12] Rand Hindi. Private smart contracts using homomorphic encryption, 2023. pages 4

[13] Uniswap docs — uniswap. pages 9, 10

[14] Hayden Adams, Noah Zinsmeister, and Dan Robinson. docs/static/whitepaper.pdf at main · uniswap/docs, 2020. pages 10

[15] How uniswap works — uniswap. pages 10, 24, 27, 30

[16] Hayden Adams. Uniswap whitepaper, 2020. pages 10

[17] Ravital Solomon. Building a truly dark dark pool, 2024. pages 12, 17, 18

[18] How dark pools quietly influence crypto markets, 10 2022. pages 12, 13

[19] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. *2015 IEEE Trustcom/BigDataSE/ISPA*, 1, 2015. pages 13

[20] Microsoft. Trusted execution environment (tee), 10 2023. pages 13

[21] Intel . Intel® software guard extensions. pages 13

[22] IEEE Digital Privacy . What is multiparty computation? - ieee digital privacy. pages 14

[23] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. Secure multi-party computation: Theory, practice and applications. *Information Sciences*, 476:357–372, 02 2019. pages 14

[24] Ethereum.org . Zero-knowledge proofs, 09 2023. pages 14, 15

[25] Lisa A . Introduction to fhe: What is fhe, how does fhe work, how is it connected to zk and mpc, what are the fhe use cases in and outside of the blockchain, etc., 2023. pages 15, 16, 17

[26] Katharina Boudgoust and Peter Scholl. Simple threshold (fully homomorphic) encryption from lwe with polynomial modulus, 2023. pages 16, 17

[27] Aayush Jain, Peter Rasmussen, and Amit Sahai. Threshold fully homomorphic encryption, 2017. pages 16

[28] Owen. fhe-darkpools, 2024. pages 17

[29] Christopher Bender and Joseph Kraut. Renegade whitepaper, 2023. pages 18

[30] Robert Annessi. Backrunning private transactions using multi-party computation — flashbots, 02 2023. pages 18, 19, 23, 61, 62

[31] Rand Hindi. What is fhevm — 0.4 — fhevm, 2024. pages 22

[32] Refactoring Guru. Design patterns, 2014. pages 24

[33] Derao. Ethereum under the hood part 3 (rlp decoding), 2021. pages 27

[34] Recursive-length prefix (rlp) serialization, 2024. pages 27

[35] bakaoh. Github - bakaoh/solidity-rlp-encode: Rlp encoding in solidity, 2018. pages 32, 47

[36] Iso 25010. pages 34

[37] Hamdi Allam. hamdiallam/solidity-rlp, 2018. pages 36

[38] Sourena Khanzadeh, Noama Samreen, and Manar H Alalfi. Optimizing gas consumption in ethereum smart contracts: Best practices and techniques. 10 2023. pages 49, 50, 51, 52

[39] ZAMA. Connecting to zama devnet — fhevm, 04 2024. pages 69

[40] ZAMA. Zama faucet, 2024. pages 69

Appendix A

User Guide

To use the backrunning solution detailed in this study, there are two necessary steps needed. First the user must connect to the ZAMA devnet and get ZAMA tokens from their faucet. Secondly, the user has to load the necessary code from GitHub and execute the code on the Remix IDE.

A.1 Connecting To The ZAMA Devnet

To connect to the ZAMA devnet, the user must first add the ZAMA network to their Metamask wallet by manually inputting the following information [39]:

- **Network Name**: Zama Network
- **New RPC URL**: `https://devnet.zama.ai`
- **Chain ID**: 9000
- **Currency symbol**: ZAMA
- **Gateway URL**: `https://gateway.devnet.zama.ai`
- **Block explorer URL (Optional)**: `https://explorer.devnet.zama.ai`

Following this, the user must then retrieve some of the devnet tokens in order to test the code. The ZAMA token can be retrieved from the ZAMA Faucet[1] [40] by entering the address of the wallet that has been connected to the ZAMA network.

A.2 Executing The Code

Once connected to the ZAMA devnet, the user must then upload the relevant source code to the Remix IDE. The source code of this solution can be found on GitHub[2].

[1] `https://faucet.zama.ai/`

[2] `https://github.com/MajedAlthonayan/FHEBackrun`

This source code must be combined with the fhEVM v0.4 “lib” folder [3] on the same directory.

Given this source code, the user can now deploy and test the code on the ZAMA devnet to successfully backrun user transactions.

[3] `https://github.com/zama-ai/fhevm/tree/release/0.4.x/lib`

Appendix B

Ethics Checklist

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		✓
Does your project involve the use of human embryos?		✓
Does your project involve the use of human foetal tissues / cells?		✓
Section 2: HUMANS		
Does your project involve human participants?		✓
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from "Human Embryos/Foetuses" i.e. Section 1)?		✓
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		✓
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		✓
Does it involve processing of genetic information?		✓
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		✓
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		✓

	Yes	No
Section 5: ANIMALS		
Does your project involve animals?		✓
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		✓
If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		✓
Could the situation in the country put the individuals taking part in the project at risk?		✓
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		✓
Does your project deal with endangered fauna and/or flora /protected areas?		✓
Does your project involve the use of elements that may cause harm to humans, including project staff?		✓
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		✓
Section 8: DUAL USE		
Does your project have the potential for military applications?		✓
Does your project have an exclusive civilian application focus?		✓
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		✓
Does your project affect current standards in military ethics – e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		✓

	Yes	No
Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?		✓
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		✓
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?		✓
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		✓
SECTION 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		✓
Will your project use or produce goods or information for which there are data protection, or other legal implications?		✓
SECTION 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		✓