COMMON: Order Book with Privacy

Albert Garreta\textsuperscript{3}, Adam Gągol\textsuperscript{1,2}, Aikaterini-Panagioti Stouka \textsuperscript{3}, Damian Straszak \textsuperscript{1,2}, and Michal Zajac \textsuperscript{3}

\textsuperscript{1}Aleph Zero Foundation
\textsuperscript{2}Cardinal Cryptography, \{adam.gagol,damian.straszak\}@cardinals.cc
\textsuperscript{3}Nethermind Research,\{albert,aikaterini-panagioti,michal\}@nethermind.io

Tuesday 5\textsuperscript{th} December, 2023

Abstract

Decentralized Finance (DeFi) has witnessed remarkable growth and innovation, with Decentralized Exchanges (DEXes) playing a pivotal role in shaping this ecosystem. As numerous DEX designs emerge, challenges such as price inefficiency and lack of user privacy continue to prevail. This paper introduces a novel DEX design, termed COMMON, that addresses these two predominant challenges. COMMON operates as an order book, natively integrated with a shielded token pool, thus providing anonymity to its users. Through the integration of zk-SNARKs, order batching, and Multiparty Computation (MPC) COMMON allows to conceal also the values in orders. This feature, paired with users never leaving the shielded pool when utilizing COMMON, provides a high level of privacy.

To enhance price efficiency, we introduce a two-stage order matching process: initially, orders are internally matched, followed by an open, permissionless Dutch Auction to present the assets to Market Makers. This design effectively enables aggregating multiple sources of liquidity as well as helps reducing the adverse effects of Maximal Extractable Value (MEV), by redirecting most of the MEV profits back to the users.

\textbf{Keywords}— private order book, privacy, DEX, ZK-SNARK, batching, Dutch auction, MPC, smart contracts, MEV, threshold encryption
Contents

1 Introduction .................................................. 4
  1.1 Our Contribution .............................................. 5
  1.2 Other Related Works ................................……… 6
  1.3 Reading the Paper ............................................ 8

2 Preliminaries .................................................. 8
  2.1 Cryptographic primitives ..................................... 8
  2.2 Arithmetic ................................................... 11
  2.3 Frontend vs Backend ......................................... 14

3 Blockchain, Smart Contracts and Data Types .............. 15
  3.1 Blockchain and Smart Contracts .......................... 15
  3.2 Constants .................................................... 17
  3.3 Common Types ............................................... 18
  3.4 Composite Types ............................................ 19

4 Technical Overview ............................................ 24
  4.1 Shielded Token Pool ......................................... 24
  4.2 COMMON – a Bird’s Eye View ............................. 26
  4.3 Order Book .................................................. 28
  4.4 Swap Engine .................................................. 31

5 Security and Privacy .......................................... 32
  5.1 Security Guarantees for Users ............................ 33
  5.2 Security Guarantees for Market Makers .................. 35
  5.3 Privacy Guarantees ........................................... 35
  5.4 Price .......................................................... 36

6 Order Book .................................................... 37
  6.1 Storage ....................................................... 37
  6.2 Calls .......................................................... 38
  6.3 User Actions ................................................... 46
  6.4 Updaters ...................................................... 52

7 Swap Engine ................................................... 53
  7.1 Storage ........................................................ 53
  7.2 Calls .......................................................... 54
  7.3 User Actions ................................................... 56
  7.4 Updaters ....................................................... 57
### 8 Relations

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Constraint bundle: Merkle tree membership</td>
<td>58</td>
</tr>
<tr>
<td>8.2 Constraint bundle: correct link between order and note</td>
<td>58</td>
</tr>
<tr>
<td>8.3 Constraint bundle: correct token nullifier</td>
<td>59</td>
</tr>
<tr>
<td>8.4 New order</td>
<td>59</td>
</tr>
<tr>
<td>8.5 Constraint bundle: Order ownership</td>
<td>60</td>
</tr>
<tr>
<td>8.6 Cancel order</td>
<td>60</td>
</tr>
<tr>
<td>8.7 Claim cancelled order</td>
<td>61</td>
</tr>
<tr>
<td>8.8 Claim swap</td>
<td>62</td>
</tr>
<tr>
<td>8.9 Deposit tokens</td>
<td>63</td>
</tr>
<tr>
<td>8.10 Withdraw tokens</td>
<td>64</td>
</tr>
<tr>
<td>8.11 Constraint bundles: non-standard arithmetics</td>
<td>65</td>
</tr>
</tbody>
</table>

### 9 Decryption Oracle

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Functionality</td>
<td>66</td>
</tr>
<tr>
<td>9.2 Decryption Oracle Instantiation Pattern</td>
<td>67</td>
</tr>
<tr>
<td>9.3 Instantiation using Threshold ElGamal Encryption</td>
<td>70</td>
</tr>
<tr>
<td>9.4 Instantiation with a Single Party using SAVER</td>
<td>74</td>
</tr>
<tr>
<td>9.5 Trusted Execution Environment Instantiation</td>
<td>76</td>
</tr>
</tbody>
</table>

### 10 Extensions and Practical Considerations

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Recovering from Decryption Oracle Malfunction</td>
<td>76</td>
</tr>
<tr>
<td>10.2 Compliance and Avoiding Bad Actors</td>
<td>77</td>
</tr>
<tr>
<td>10.3 Correlations Based on Transaction Origin</td>
<td>77</td>
</tr>
<tr>
<td>10.4 Correlations Based on Token Amounts</td>
<td>78</td>
</tr>
<tr>
<td>10.5 Adding Noise to Hide Order Direction</td>
<td>79</td>
</tr>
</tbody>
</table>

### A Precision of Fixed Point

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Precision of Fixed Point</td>
<td>79</td>
</tr>
</tbody>
</table>
1 Introduction

Decentralized Finance (DeFi) has sparked a wave of financial innovation, elevating decentralization, trustlessness, and accessibility to the forefront of the global finance landscape. Central to this paradigm shift are Decentralized Exchanges (DEXes), which serve as the bedrock of DeFi on blockchain platforms. In recent years, the DeFi landscape has witnessed an explosion of creativity and experimentation, giving rise to a multitude of DEX designs. These innovations, including Constant Function Market Makers (CFMMs \cite{AZR, AC20}), Concentrated Liquidity models \cite{AZS+21}, Order Book-based DEXes \cite{WBI7}, and DEX aggregators, all share a common goal: enhancing the user experience and bridging the gap between decentralized and centralized exchanges, all while preserving users’ control over their assets. However, amidst this notable advancement, two persistent challenges remain largely unaddressed:

- **Price inefficiency**: the fragmentation of liquidity across DEXes continue to hinder optimal pricing for users in decentralized exchanges. Moreover, the risk (or rather certainty) of failing victim to various MEV extraction tactics, such as front-running and sandwich attacks \cite{ZQT+21, EMC19, WZY+22, QZG22, ZXE+23}, considerably exacerbates the appeal of centralized exchanges for large-scale trading.

- **Lack of Privacy**: the open nature of blockchain transactions, which allows anyone to inspect transaction records of any blockchain user poses a significant privacy concern \cite{AKR+13, CELR18}. For this reason many users prefer centralized exchanges, where they can, at the very least, hide their trading activities from other users and general public, albeit at the cost of ceding control to the exchange. This preference for privacy has contributed to the substantial liquidity disparity between decentralized and centralized exchanges.

Addressing the issue of price inefficiency remains an ongoing research topic within the blockchain community. To combat liquidity fragmentation, various strategies have been proposed, notably the use of DEX aggregators (such as 1inch) for consolidating liquidity from on-chain sources and Liquidity Networks for cross-chain liquidity. Some modern DEX designs (such as Cow or Uniswap X) incorporate built-in auction systems to bolster liquidity. Still, all these designs are to some extent susceptible to increasingly sophisticated MEV extraction strategies. In fact, it is widely acknowledged that MEV represents an inherent facet of blockchain ecosystems and, as such, cannot be eliminated outright. Instead, the most effective approach to mitigate MEV’s adverse effects \cite{DGK+19} appears to lie in designing DeFi protocols to be *MEV-aware*. This involves implementing mechanisms that systematically capture significant portions of what would otherwise be MEV bot profits \cite{CK22}, redirecting these returns into the protocol for the benefit of its users.

Enhancing privacy for DEX users, as well as blockchain users in general, has been a focal point of research since the inception of blockchain technology (see \cite{BCDF23} for a survey). Initially, pseudonymity provided by assets like Bitcoin was believed to guarantee privacy, but this claim was debunked early on \cite{AKR+13, CELR18}. In the seminal work
on ZeroCoin [MGG13] (and later ZCash [HBHW22]) the notion of shielded pools has been introduced as a tool for improving user privacy. They have been built using a novel (at that time) and powerful cryptographic tool: zk-SNARKS (see Definition 2.2). The main idea of shielded pools is to allow users to deposit coins in a "magic box" where the coins of different users are mixed, and the link between deposits and withdrawals is obscured. This idea has since been refined and generalized by many different protocols (such as TornadoCash [PSS19], Aztec [Wil18], Namada, Nocturne [Lab23], Zswap [EKKV22] or Privacy Pools [BIN+23]).

Especially in the recent years, along with the breakthroughs in zero-knowledge proofs [BGKS20, GWC19, BGH19, CHM+19, BSCR+18], one could witness increased interest in technologies based on ZKPs. Nonetheless, currently deployed privacy solutions are often confined to coin mixing in shielded pools and lack seamless integration with DeFi, requiring users to withdraw from these pools to engage with specific protocols.

Efforts have also been made to design private DEXes using ZKPs. For instance, developing an anonymous version of a Constant Function Market Maker (CFMM), akin to DEXes like Uniswap, turns out to be possible – in such a DEX, swaps are public, yet user identities remain concealed. There are, however, inherent obstacles in extending the privacy further, particularly in ensuring that swap values are also kept confidential. For instance Angeris, Evans and Chitra in [AEC21, CAE22] give strong impossibility results regarding privacy of the state in such contracts. In the same work, transaction batching is proposed as a way to circumvent these limitations (see [JDE+23] and [cow pen and]).

Another approach is to build DEXes based on peer-to-peer trading, yet these suffer from user experience issues: the users need to involve in complex counterparty selection protocols, which makes order matching quite time-consuming but also requires the users to stay online for long periods of time. More generally, all DEXes based purely on ZKPs suffer from a common issue: every part of the DEX’s state must be known to at least one party, which surprisingly makes it hard or even impossible to design protocols where also the traded values are masked. One could argue that to achieve such strong privacy more sophisticated tools such as Multiparty Computation (MPC) [Yao82, GMW87, BG89] or Trusted Execution Environments (TEE) [SAB15] are necessary. For more details regarding related works cf. Subsection 1.2.

1.1 Our Contribution

This paper introduces COMMON, a novel Decentralized Exchange design, with the primary goal of tackling the two issues mentioned earlier: price inefficiency and lack of privacy. COMMON functions as an order book providing high privacy guarantees, offering users both

---

1 In the blockchain community the term "zero-knowledge proofs" (ZKPs) is often colloquially used as a synonym for zk-SNARKs. In this part of the paper we follow this (slightly incorrect) convention.

2 Simplest way to achieve it: hold tokens in a shielded pool and then an anonymous swap boils down to the following sequence of actions: unshield to a one-time account, swap tokens, and shield back.

3 This is primarily because the "prover" party in the ZKP must know the state it is proving something about. We also make the silent assumption here that the prover is a single party, and not a multi-party committee.
spot trade execution and the option to set (possibly long-living) limit orders. In addition to internal order matching within COMMON, the order matching engine leverages external liquidity sources to attain optimal prices.

The privacy features of COMMON are achieved using zk-SNARKs and a generic cryptographic primitive called a Decryption Oracle that we introduce as part of the design. In Section 9 we detail how to instantiate the Decryption Oracle using Multi-party Computation. Technically, while zk-SNARKs are used to achieve anonymity, the Decryption Oracle allows to achieve a level of confidentiality over the amounts in users’ trades. As previously remarked, this requires techniques beyond ZKPs in order to achieve a solution that doesn’t rely on trust to a single entity, hence the use of Multi-party Computation.

COMMON natively integrates with a shielded token pool, ensuring user anonymity. Importantly, placing orders in COMMON does not force the user to exit the shielded pool, which with the help of the Decryption Oracle allows users to maintain the confidentiality of their orders. The aggregated value, a combination of multiple orders, is only disclosed at the time of order matching. This not only enhances privacy, as opposed to mere anonymity, but also serves as a MEV-protection mechanism. Indeed, prior to revealing the aggregated value, the batch of orders is sealed to prevent specialized actors from injecting strategic orders into it. Subsequently, a separate component of COMMON: the SWAP-ENGINE takes over and tries to trade the batch in an optimal way.

The SWAP-ENGINE is designed in a MEV-aware fashion and works in two phases: 1) Internal Matching – the users’ orders are directly matched incurring no fee, 2) Dutch Auction – the remaining funds are sold in a public auction. The auction starts from a relatively unattractive price, gradually reduces it, block by block, to encourage Market Makers, to buy and/or arbitrage with respect to other markets. This mechanism allows to effectively aggregate all the existing on-chain and off-chain liquidity (CEXes and other chains, via bridges). Importantly, it achieves this in a manner that directs a significant portion of MEV profits back to COMMON’s users.

1.2 Other Related Works

Processing orders in frequent batches has been suggested by [EPJ15] as a better market design response compared to continuous limit order books, because it prevents wasteful race for tiny advantages in speed. Moreover, batching of transactions for enhancing privacy was presented in [AEC21] and later assessed under a simple adversarial model in [CAE22]. As already mentioned, some examples of other protocols that use the technique of batching are [cow] and [pen].

The closest to our work is ZSwap protocol proposed by Penumbra [pen]. ZSwap is based on [JDE+23] and includes batching of encrypted orders (using homomorphic threshold encryption), decryption of the aggregated amounts and “clearing” trades via routing across multiple public concentrated liquidity positions that can open and close in the same transaction anonymously. Note that in our protocol the SWAP-ENGINE smart

\[\text{4}\text{Apart from that we also show how, alternatively, a Decryption Oracle can be implemented using Trusted Execution Environments (TEEs) or just a single trusted party.} \]
contract performs internal matching as much as possible of the aggregated decrypted amounts and then the excessive amount is not traded against public concentrated liquidity positions. Instead, our protocol can attract liquidity from many sources (DEXes, CEXes) via Market Makers who compete to buy the excessive amount in a Dutch auction. Some examples of other protocols that utilize Dutch auction to attract liquidity are [ca23, im].

Examples of other protocols that use the homomorphic property or aggregation to ensure privacy are [DDD+23, SWA23, EKKV22, Fla22, EMP+21]. Zama protocol [DDD+23] uses homomorphic encryption to ensure privacy in smart contracts and SWA23 proposes a framework that uses fully homomorphic encryption to support privacy preserving smart contracts. In EKKV22 sparse homomorphic commitments are used to merge transactions privately. Suave [Fla22] facilitates a Universal Preference Environment where users can submit their encrypted preferences that will be aggregated by block builders and will form a part of a block. EMP+21 enables private exchange of tokens via an aggregatable signature scheme.

Another technique that has been used in DEXs to enhance privacy is combining Zero-knowledge proofs (ZKP) and Multiparty Computation (MPC). Some examples of works that use this technique are [BDF21, GY21, BK, ano]. BDF21 uses a publicly verifiable MPC protocol to match secret-shared orders. In GY21 there is (i) a bidding phase, where the users send commitments to their orders to a smart contract, and (ii) a reveal phase, where the users send encryptions of their orders to the same smart contract. The encryptions use the public key of an operator (or MPC) who later can collect the encrypted orders, decrypt them and match them. In Renegade BK relayers manage users’ wallets (this means that they are able to view the unencrypted wallet that they control) and they run MPC in order to match orders. BK also uses collaborative zk-SNARKS [OB22] (no party knows the whole secret) for proving that the MPC computations were performed correctly. ano facilitates an Intent-Based Execution where intents are gossiped in a peer to peer network, they are matched by solvers using MPC or other method and finally they get included on chain. In our work, to make it as practical as possible, we avoid extensive use of generic MPC, and only apply it for decryption of aggregated amounts, not for the order matching; the order matching is performed in plaintext by the SWAP-ENGINE smart contract.

Some DEX protocols use the commit-reveal technique, where the users send initially commitments to their orders, and reveal them afterwards when all the orders have been collected. Some examples are [BDF21, CC21]. Another commit-reveal protocol that also uses ZK membership proofs to hide the identity of the users before the reveal phase is MDF22. Moreover, MDF22 in order to prevent Miner Extractable Value (MEV) (see [ISSW22] for its formal definition) proposes also a width-sensitive frequent batch auction (WSFBA) which is an improvement to FBA [EPJ15], because it provides the same guarantees but even under a monopolistic Market Maker and without demanding from the clients to submit a limit order. Note that in our case the commit-reveal technique is not suitable because we would like to hide the value of the orders not only before the settling of the orders but even after. We note that even though in our protocol a user
reveals the direction, this can be mitigated by generating fake orders that encrypt “0” – see Subsection 10.5. Furthermore, compared to [MDFO22], due to the utilization of MPC for decryption, we do not ask the users (i) to participate again and reveal their order (ii) to put an escrow that will be removed if they do not reveal their order. Also after the internal matching via geometric mean (which was also suggested by [MDFO22]), as already mentioned, we use extra liquidity sources from external Market Makers via a Dutch auction.

Since in our protocol we perform both order collection and order matching on-chain, front-running protection protocols, such as [CIM+22], cannot be applied, as they assume secure communication between traders and Market Makers.

1.3 Reading the Paper

The recommended order of reading the paper is to start with the Introduction (Section 1) and then go straight to the Technical Overview (Section 4). The technical overview refers to other sections but is largely self-contained and is meant to provide the reader with a high-level understanding of COMMON. Subsequently the reader is encouraged to dive into the details: Order Book (Section 6), Swap Engine (Section 7), Decryption Oracle (Section 9) and zk-SNARK Relations (Section 8). The material in Section 2 provides some preliminaries on cryptography, zk-SNARKS, and arithmetic in this paper. Section 3 gives preliminaries on blockchain and smart contracts, as well as introduces data types used in COMMON. Section 4 provides an informal discussion of the guarantees provided by COMMON in terms of security and privacy and discusses which will be the price of trading in a best and worst case scenario. Finally, Section 10 discusses practical aspects of implementing COMMON, including various extensions and improvements that were left out of scope of this paper.

2 Preliminaries

2.1 Cryptographic primitives

2.1.1 zk-SNARKs

Next we formally define the zk-SNARK cryptographic primitive, and afterward we make some conventions specific to our use-case.

**Definition 2.1** (Relation). An indexed relation \( R \) is a set of triples \((idx, x; w) \in \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^*\) consisting of an index \( idx \), an input \( x \) (also called statement or instance), and a witness \( w \). Intuitively, \( idx \) represents parameters like a finite field.

A function \( \kappa : \mathbb{N} \to \mathbb{R} \) is called *negligible* if for all \( c > 0 \) there exists \( x_c \geq 0 \) such that \( \kappa(x) \leq x^{-c} \) for all \( x \geq x_c \). On the other hand, \( \kappa \) is called *sublinear* if \( \kappa(x) = o(x) \), i.e. if \( \lim_{x \to \infty} \kappa(x)/x = 0 \).

\(^5\) As is common in the literature we occasionally use the notation \( R(idx, x; w) \) to mean \((idx, x; w) \in R\).
Definition 2.2 (Universal SNARK with preprocessing). A universal Succinct Non-Interactive Argument of Knowledge (SNARK) with preprocessing for an indexed relation \( R \) is a tuple of algorithms \( \Pi = (G, \text{Ind}, P, V) \) such that:

- \( G \) is a Probabilistic Polynomial Time (PPT) algorithm that takes as input \( 1^\lambda \) and a size bound \( N \in \mathbb{N} \), and outputs a structured reference string \( \text{sr} \). Here \( \lambda \) is a security parameter.
- \( \text{Ind} \) is a deterministic polynomial time algorithm that receives \( \text{sr} \) and an index \( \text{id} \) of size at most \( N \), and outputs verifier and prover parameters \( \text{vp}, \text{pp} \).
- \( P \) is a PPT algorithm that receives \( (\text{pp}, \text{x}, w) \) as input, and outputs a string of bits \( \pi \), called proof.
- \( V \) is a PPT algorithm that receives \( (\text{vp}, \text{x}, \pi) \) as input, and outputs accept or reject.

We assume both \( G \) and \( \text{Ind} \) are run by a trusted party. We require the following properties to hold:

- **Perfect completeness.** For all \( \lambda \geq 0 \), \( N \in \mathbb{N} \), \( (\text{id}, \text{x}, w) \in R \) with \( |\text{id}| \leq N \), \( \text{sr} \leftarrow G(1^\lambda, N) \), \( \text{pp}, \text{vp} \leftarrow \text{Ind}(\text{sr}, \text{id}) \) and \( \pi \leftarrow P(\text{pp}, \text{x}, w) \), we have accept \( \leftarrow V(\text{vp}, \text{x}, \pi) \) with probability 1.
- **Knowledge soundness (adaptive).** For all \( \lambda \geq 0 \) and \( N \in \mathbb{N} \) and PPT algorithms \( A_1, A_2 \) there exists a PPT algorithm \( \text{Ext} \), called extractor, such that

\[
\Pr \left[ \begin{array}{c}
(\text{id}, \text{x}, w) \notin R \\
|\text{id}| \leq N \\
\langle A_2(\text{st}), V(\text{vp}, \text{x}) \rangle = 1
\end{array} \right] \frac{\text{sr} \leftarrow G(1^\lambda, N)}{(\text{id}, \text{x}, \text{st}) \leftarrow A_1(\text{sr})} \frac{\text{w} \leftarrow \text{Ext}(\text{sr})}{(\text{pp}, \text{vp}) \leftarrow \text{Ind}(\text{sr}, \text{id})} = \kappa(\lambda),
\]

where \( \kappa : \mathbb{N} \to [0, 1] \) is a negligible function, and \( \langle A_2(\text{st}), V(\text{vp}, \text{x}) \rangle = 1 \) means that \( V(\text{vp}, \text{x}) \) outputs accept after interacting with \( A_2(\text{st}) \).

- **Succinctness.** For all \( \lambda \geq 0 \), \( N \in \mathbb{N} \), \( \text{sr} \leftarrow G(1^\lambda, N) \), \( (\text{id}, \text{x}, w) \in R \) with \( |\text{id}| \leq N \), \( (\text{pp}, \text{vp}) \leftarrow \text{Ind}(\text{sr}, \text{id}) \) and \( \pi \leftarrow P(\text{pp}, \text{x}, w) \), the proof \( \pi \) has size \( \text{poly}(\lambda + |\text{x}|) \). Moreover, \( V(\text{vp}, \text{x}, \pi) \) runs in time \( \text{poly}(\lambda + |\text{x}|) \).
- **Zero-knowledge.** There exists a PPT algorithm \( \text{Sim} \), called simulator, such that for every PPT algorithms \( (A_1, A_2) \) it holds that

\[
\Pr \left[ \begin{array}{c}
(\text{id}, \text{x}, w) \in R \\
|\text{id}| \leq N \\
\langle \text{pp}(\text{x}, \text{w}), A_2(\text{st}) \rangle = 1
\end{array} \right] \frac{\text{sr} \leftarrow G(1^\lambda, N)}{(\text{id}, \text{x}, \text{w}, \text{st}) \leftarrow A_1(\text{sr})} = 
\Pr \left[ \begin{array}{c}
(\text{id}, \text{x}, w) \in R \\
|\text{id}| \leq N \\
\langle \text{Sim}(\text{trap}, \text{id}, \text{x}), A_2(\text{st}) \rangle = 1
\end{array} \right] \frac{(\text{sr}, \text{trap}) \leftarrow \text{Sim}(1^\lambda, N)}{(\text{id}, \text{x}, \text{w}, \text{st}) \leftarrow A_1(\text{sr})}.
\]
Convention 2.3. All our indexed relations $R$ will be such that if $(\text{idx}, x, w) \in R$, then $\text{idx} = (p, n, m)$, where:

- $p$ is a prime of $\text{poly}(\lambda)$ bits, and $n, m$ are two nonnegative integers.
- $x$ represents an $n$-tuple of elements from the finite field $\mathbb{F}_p$.
- $w$ represents a $m$-tuple of elements from $\mathbb{F}_p$.

Because of this, we will often omit referring to $\text{idx}$ and will simply speak of input-witness pairs $(x, w) \in \mathbb{F}_p^n \times \mathbb{F}_p^m$. Moreover, we will denote $\mathbb{F}_p$ by $\mathbb{F}$.

Additionally, and for ease of presentation, we will also omit referring to the algorithms $G$ and $\text{Ind}$ further, as well as to the proving and verifier parameters $pp, vp \leftarrow \text{Ind}(\text{srs, idx})$. We instead assume these are generated and fixed at the initialization of COMMON and used whenever appropriate.

2.1.2 Scalar Fields

In COMMON we often deal with token amounts and prices and would ideally want these values to be represented by single $\mathbb{F}$ elements and the arithmetic on $\mathbb{F}$ to be compatible with regular integer arithmetic (see also Subsection 2.2). For this reason, it is the most practical to have $\mathbb{F} = \mathbb{F}_p$ with $p$ quite large, for instance $p \approx 2^{256}$, as is the case when the proof system is instantiated with the BLS12-381 pairing system [BLS02].

For the sake of concreteness, in this paper we fix $\mathbb{F}$ to be the scalar field of BLS12 − 381 and thus $p = 0x73eda753299d7d483339d80809a1d80553ba402fffe5bfeffffffff00000001$.

We also define the type Scalar to hold elements of the field $\mathbb{F}$ (as introduced in Section 2.1.2) with the standard field arithmetic defined over elements of Scalar.

2.1.3 Choice of Proof Systems

COMMON is generic over the choice of a particular zk-SNARK – any can be used as long as it satisfies the definition in Subsection 2.1.1. However, the choice of a particular proof system has crucial impact on the overall efficiency:

- **Prover time**: end-users are intended to generate proofs, hence the prover efficiency plays an important role on the user experience.

- **Verifier time**: proofs are verified on-chain in a smart contract, thus the verifier time heavily impacts the gas-efficiency of the solution. The existence of particular precompiles (EVM) or host functions (substrate) for given elliptic curves and pairing systems might also be of crucial importance when choosing the proof system.

- **Proof size**: the proof needs to be included in a transaction (calldata in EVM), but does not need to be recorded in the state. This incurs some constraints on the proof size, but the verifier time seems to have more impact.

Given the above constraints, at the time of writing, the most promising choices seem to be Groth16 [Gro16a] and PLONK [GWC19] proof systems.
2.1.4 Hashing

We make use of a cryptographic hash function $\text{Hash} : F^n \rightarrow F$ for not too large $n$ (say $n \leq 10$). In practice, they can be instantiated with Poseidon [GKR+21]. Whenever we write $\text{Hash}(x)$ and $x$ is not a tuple of $F$ in an obvious way, we assume that there is a generic, deterministic mapping between the type of $x$ and $F^n$ for some constant $n$.

2.1.5 Encryption scheme

We let $(KGen, Enc, Dec)$ be an encryption scheme, and we let $(pk, sk) \leftarrow KGen(1^\lambda)$ be a pair of keys for this encryption scheme.

**Definition 2.4.** Let $E$ be an encryption scheme. We say that $E$ is additively homomorphic if, for any ciphertexts $c, c'$ encrypted under public key $pk$ with a corresponding secret key $sk$, holds

$$E.\text{Dec}(pp, sk, c) + E.\text{Dec}(pp, sk, c') = E.\text{Dec}(pp, sk, c + c').$$

In Section 9.2 we introduce (chunked) ElGamal as an example of such an encryption schemes.

2.2 Arithmetic

In the description of COMMON, we use several numeric types, such as FixedPoint, Amount, Ratio and others, representing rational or integer numbers with various bounds. Apart from being involved in arithmetic operations executed in smart contracts, values of these types also appear in hashing (see Section 2.1.4) or in relations for zk-SNARKS, which take field elements as input (see Section 8) thus need to be expressible as $F$ elements.

Below we describe all the numeric types used in COMMON making sure that in each case under the hood they are represented by integers in the range $[0, \sqrt{p})$ which guarantees that they can be canonically mapped to Scalar in a way that multiplying two such field elements still does not exceed $p$ (no carry-over). This is a technical requirement that is used in Section 8 in order to correctly implement relations validating "regular arithmetic" using native field arithmetic in $F$. We note that conversions between "normal numeric types" and Scalar are most susceptible to bugs and logical errors in implementations often leading to underconstrained circuits, thus it is highly recommended to explicitly check bounds in the circuit for each value which represents a "bounded type".

2.2.1 Type FixedPoint

For the sake of completeness in this section we define the fixed-point arithmetic. While our definitions are standard, we emphasize in a few places how our choice of parameters makes this arithmetic compatible with the native arithmetic in $F$ and explain what does it precisely mean.
We fix a precision scalar $M$ and bound scalar $B$ (importantly, we assume $M \leq B < \sqrt{p}$) and define the type $\text{FixedPoint}(M, B)$ to be the set of integers $[0, B - 1]$. Conceptually, an element $y \in \text{FixedPoint}(M, B)$ represents the rational number $\frac{y}{M}$.

From now on, we drop the references to $M$, $B$ from the notation $\text{FixedPoint}(M, B)$ and write simply $\text{FixedPoint}$.

**Addition:** the sum $+: \text{FixedPoint} \times \text{FixedPoint} \rightarrow \text{FixedPoint} \cup \{\text{Err}\}$ is defined as follows:

$$y_1 + y_2 = \begin{cases} y_1 + y_2 & \text{if } y_1 + y_2 < B, \\ \text{Err} & \text{otherwise} \end{cases}.$$

Note importantly that addition can fail, in case we exceed the range (denoted by $\text{Err}$)

**Multiplication:** similarly, we define the multiplication

$$y_1 \cdot y_2 = \begin{cases} \lfloor \frac{y_1 y_2}{M} \rfloor & \text{if } y_1 y_2 < B \cdot M, \\ \text{Err} & \text{otherwise} \end{cases}.$$

Lemma A.1 in Appendix A states that the above defined arithmetic agrees with regular rational arithmetic, up to perhaps a small error $\frac{1}{M}$.

It is also important to note that the above definitions are 100% compatible with the relations $\text{CONSTR}_{\text{FixedPtAdd}}$ and $\text{CONSTR}_{\text{FixedPtMul}}$ that are defined in Section 8.11 and take triples of $F$ inputs. More specifically, if $y_1, y_2, y_3 \in \text{FixedPoint}$ then all these elements are naturally also elements of $F$ and it holds:

$$y_1 + y_2 = y_3 \iff \text{CONSTR}_{\text{FixedPtAdd}}(y_1, y_2, y_3),$$
$$y_1 \cdot y_2 = y_3 \iff \text{CONSTR}_{\text{FixedPtMul}}(y_1, y_2, y_3).$$

**Division:** occasionally we also need to divide $\text{FixedPoint}$ values. To this end, we define $/: \text{FixedPoint} \times \text{FixedPoint} \rightarrow \text{FixedPoint} \cup \{\text{Err}\}$

$$y_1 / y_2 = \begin{cases} \left\lfloor \frac{y_1 - M y_2}{y_2} \right\rfloor & \text{if } \left\lfloor \frac{y_1 - M y_2}{y_2} \right\rfloor < B \\ \text{Err} & \text{otherwise} \end{cases}.$$ where the operations on the right-hand side are rational number operations. In other words, $y_1/y_2$ is the result of making the rational number division $y_1/y_2$, removing decimals beyond the $M$-th decimal, and multiplying by $M$ so that the resulting value has no decimals.

---

6 The main idea behind this assumption is to allow performing some arithmetic operations in $F$ without wrap-arounds (going beyond $p$).
2.2.2 Type Amount

The type `Amount` is meant to hold integer values in the range \([0, \text{MAXSUPPLY}]\) representing token amounts, where \(\text{MAXSUPPLY}\) is a constant defined in Section 3.1.4. The constants are selected so as to satisfy \(\text{MAXSUPPLY} < \frac{B}{M}\).

**Division of Amounts:** in the protocol we occasionally need to divide amounts in order to obtain prices (see Section 2.2.3 for the definition of Price). In order to compute \(\frac{a_1}{a_2}\) for \(a_1, a_2 \in \text{Amount}\) we just represent both these amounts as values \(y_1, y_2\) of type `FixedPoint`, where \(y_1 = a_1 \cdot M\) and \(y_2 = a_2 \cdot M\), and compute the quotient \(y_1/y_2\) as defined in Section 2.2.1.

2.2.3 Types Ratio and Price

The type `Price` is simply an alias for `FixedPoint`, however it is meant to represent prices of tokens.

The type `Ratio` is also the same as `FixedPoint` under the hood, but we also enforce that the rational value it corresponds to is in the interval \([0, 1]\).

**Multiplying Amount by Price:** when computing swap results we need to compute a product of \(a : \text{Amount}\) and \(y : \text{Price}\) and get a result of type `Amount`. We thus define \(\cdot : \text{Amount} \times \text{Price} \rightarrow \text{Amount} \cup \{\text{Err}\}\)

\[
a \cdot y = \begin{cases} \lfloor \frac{a \cdot y}{M} \rfloor & \text{if } a \cdot y < B \\ \text{Err} & \text{otherwise} \end{cases}
\]

As with the previous operations involving values of type `FixedPoint`, the above multiplication definition is compatible with the relation \(\text{CONSTR}_{\text{FixedPointAmountMul}}\) from Section 8.11.2. More specifically, if \(a, y' \in \text{Amount}\) and \(y \in \text{Price}\) then all these elements are naturally also elements of \(F\) and it holds that

\[a \cdot y = y' \iff \text{CONSTR}_{\text{FixedPointAmountMul}}(a, y, y').\]

2.2.4 Types ScaledAmount and ScaledRatio

Notice that the operations \(\cdot\) and \(+\) defined on `FixedPoint` previously involve the “fractional floor” function \(\lfloor \cdot \rfloor\). This means that, if we want to, say, multiply a token amount (a value of type `Amount`) by a fraction (a value of type `Ratio`), then we need to apply the fractional floor function to some value. This is not really friendly towards the native arithmetic in \(F\); indeed, the arithmetic circuits for \(\text{CONSTR}_{\text{FixedPtAdd}}\) and \(\text{CONSTR}_{\text{FixedPtMul}}\) (cf. Section 8.11.2) have to involve complex range proofs and require roughly \(\log(p)\) gates.

However, at some specific points of our system, we would like to use additive homomorphic encryption: multiplying a value \(y\) of type `Ratio` with the encryption of a value \(a\) of type `Amount`. In general, this will not result in the encryption of the value \(y \cdot a\),
precisely because \( y \cdot a \) requires using the fractional floor function. However, in the special case where \( a \) is divisible by \( M \), say \( a = M \cdot a' \) for some \( a' \), we have:

\[
\frac{y}{M} a = ya'.
\]

Thus, if \( a = M \cdot a' \) and if we store \( a' \) instead of \( a \), we can define \( y \cdot a' \) simply as the scalar multiplication of \( y \) and \( a' \) (or \( \text{Err} \) if \( ya \) is larger than \( \text{MAXSUPPLY} \)). The resulting element \( ya' \) is given the type \( \text{Amount} \) because of constraints coming from the decryption procedure in the Decryption Oracle.

With this in mind, we introduce a new constant \( \frac{7}{N} \) and two new types:

- \( \text{ScaledAmount} \) – at the low level it holds an integer \( a \in \left[0, \frac{\text{MAXSUPPLY}}{N}\right] \), but conceptually it represents the integer value \( a \cdot N \).
- \( \text{ScaledRatio} \) – at the low level it holds an integer \( y \in [0, N] \), but conceptually it represents the ratio \( \frac{y}{N} \).

What is important about the above two types is that if we have \( a \in \text{ScaledAmount} \) and \( y \in \text{ScaledRatio} \) then both the low level, and "conceptual" representations have the same product. Indeed:

\[
(a \cdot N) \cdot \frac{y}{N} = a \cdot y.
\]

For completeness we define the product operation: \( \cdot : \text{ScaledAmount} \times \text{ScaledRatio} \to \text{Amount} \cup \{\text{Err}\} \)

\[
a \cdot y = \begin{cases} 
  ya & \text{if } ya \leq \text{MAXSUPPLY} \\
  \text{Err} & \text{otherwise.}
\end{cases}
\]

### 2.3 Frontend vs Backend

In this Section we have described the low-level "backend" representation of numeric types and have defined how arithmetic is defined. In the remaining sections of the paper (with the exception, perhaps, of Section 8) we will not look at these low-level details anymore and treat all these numbers as if they were integers or rational numbers. Whenever we perform arithmetic on values of such types we mean that they should be performed precisely as defined in Section 2.2 and thanks to strict typing there should never be ambiguity in the expressions. However, for conceptual understanding of \text{COMMON} it is best to ignore such details and just think that all operations are performed exactly, without errors, and there is no issue with converting rational numbers to \( F \) elements.

---

7 The reason for introducing a new constant here, instead of reusing \( M \), is twofold: 1) we want to have a clear distinction between \( \text{ScaledAmount} \), \( \text{ScaledRatio} \) and \( \text{Amount} \) and \( \text{Ratio} \), 2) \( M = N \) is not possible because we need \( N \) to be reasonably small.
3 Blockchain, Smart Contracts and Data Types

3.1 Blockchain and Smart Contracts

We assume the standard model of a blockchain with deterministic finality:

- blocks keep being created and finalized (liveness),
- finalized blocks are never reverted (safety),
- users can observe blocks and reliably check if they are finalized (using finality proofs),
- users can query the state at each block (perhaps using Merkle state proofs),
- user can send transaction to the chain and are not censored.

Whenever we mention waiting for a "transaction to be processed" we mean waiting until the transaction enters a block and the block is finalized. This way it is guaranteed that such a transaction cannot be reverted.

For COMMON to be efficient the blockchain should ideally have a short block-time and near-instant finality. These properties are useful for obtaining good price efficiency of the DEX. If the block time is large, then the resolution in the Dutch Auction (see Section 7) cannot be made properly optimized.

3.1.1 Smart Contracts

We require the blockchain to support Turing-complete smart contracts. This could be either EVM (Ethereum, Polygon, etc.) or WASM (like Aleph Zero) or any other. In fact, COMMON does not require any specific, non-standard chain functionality, and can be deployed merely as a system of two smart contracts. That being said, it is best if the blockchain’s runtime supports specific cryptographic precompiles (host functions) that allow for cheap zk-SNARK verification and additively-homomorphic encryption (for instance ElGamal), because that allows to make COMMON cheap in terms of gas cost.

We assume that the contract has access to environmental calls, such as:

- `currentBlock` – returns the current block number,
- `caller` – returns the contract caller (either the user or another contract).

3.1.2 Smart Contract Execution

We assume a standard model of a smart contract:

- **Code**: the logic of the contract. It consists of a number of contract calls. Each of these calls can be triggered by any user using a transaction, or by any other contract via a cross-contract call. In the pseudocode we refer to these as **Public Calls**, in contrast to **Internal Calls** which are not possible to call from outside, only by the contract itself. The purpose of internal calls is to organize the contract logic in an orderly fashion.
• **Storage:** the data the contract stores and has exclusive right to modify. We assume that there is random access to key-value collections (Map in the pseudocode) by key, and the gas cost is constant (as in EVM).

The execution cost is measured in gas units, and there are limits on maximum amount of gas per transaction, thus in particular it’s not possible to run arbitrarily long loops in contract calls, and each single call should have a known upper bound on the amount of gas it consumes.

Whenever the execution of a contract encounters an error, such as when accessing a non-existent key in a map, or failing an Assert, then the execution stops and all storage changes that resulted from the current call are reverted, as it it never happened (of course, the gas fee is still charged).

### 3.1.3 Price Oracle

In COMMON we make use of a price oracle, thus we assume that there exists a contract PRICE-ORACLE and its QueryPrices() call allows to get current prices of all tokens of interest. We strongly emphasize that:

- the use of the price oracle in COMMON is not really crucial, and an on-chain price discovery would also be fine,
- even if the oracle is ever compromised and keeps providing incorrect prices, this does not lead to the loss of any funds on COMMON. That is because the oracle is used only to improve matching efficiency, and has no effect on security.

### 3.1.4 Tokens

The tokens that can be input into COMMON and shielded are assumed to satisfy some standard interface like ERC20 (or PSP22 for substrate chains). The exact mechanics of transferring tokens are not of importance and thus in the pseudocode we are informal by using statements of the form "tokens are transferred along the transaction". In a real implementation one should replace them by first setting allowance and then using the transfer_from call. However, we choose to ignore this low-level details to improve clarity.

While the exact mechanics of the tokens are not significant, another seemingly technical and low-level aspect turns out to be quite important for COMMON, namely the token supply and its number of "decimals". Because of technical reasons: 1) efficiency of zk-SNARK proof generation, and 2) efficiency of the additively homomorphic encryption scheme, we need to avoid dealing with numbers that are too big. Specifically it is of much help if there is a universal upper bound on the maximum supply of each token – we call it MAXSUPPLY and assume that total_supply() ≤ MAXSUPPLY for each token that is traded on COMMON. For instance, a reasonable setting for MAXSUPPLY would be $10^{36}$, based on the field size $p$, and other parameters that need to be set (see Section 3.2).

Another technical issue is related to prices. We represent them with a fixed precision of M (see Section 2.2 for the description of FixedPoint) and thus require the price to
be in a specific range, in order to represent it with enough precision. This in turn brings us to the quite low level consideration of token decimals. For instance AZERO has 12 decimals, which means that 1 AZERO is actually represented as an integer $10^{12}$ and $10^{-12}$ is the smallest indivisible portion of AZERO. Similarly ETH has 18 decimals, thus 1 ETH is $10^{18}$ units (wei). Each token might have a different number of decimals, typically 6, 9, 12, 18 or 24, also each token has a different absolute value (say in USD). This can lead to anomalous cases that, e.g., a price of token A w.r.t. token B is $10^{-40}$, which might be a value not easily represented as FixedPoint (depending on the choice of $M$). For this reason we make an assumption (and below we explain how can we achieve it in practice) that a particular fixed amount of each token, say $10^D$ (for a constant $D$, think $D = 18$) should have an absolute value in the range $[10^{-K}, 10^K]$ USD, for some constant $K$. If this is satisfied, then the each price is in the range $[10^{-2K}, 10^{2K}]$ and hence we can set the precision parameter $M$ accordingly and guarantee that the rounding errors are never significant.

To solve the above two problems of max supply, and price precision, we use a simple trick: rescale the token amounts for the internal use of COMMON. The simplest way to think about it, is that for each token $T$ a constant multiplier $m_T$ is selected upon registering $T$ in COMMON so that the USD value of $10^D \cdot m_T$ units of token $T$ is roughly 1 USD. This allows us to think that each token has $D$ decimals, and that $10^D$ of each token has roughly price 1 USD. We emphasize that the multiplier $m_T$ is used only for the internal representation of the tokens, and whenever COMMON interacts with external contracts, it uses the multiplier to translate internal amounts to actual amounts.

We emphasize that the constraint that each token value $v$ is at most MAXSUPPLY (after rescaling by $m_T$) is a hard constraint, and the soundness of various zk-SNARK proofs relies on this assumption. For this reason a practical implementation of COMMON must implement a suitable safeguard that the total holdings of each token $T$ must be at most MAXSUPPLY at every time. While this cannot happen in normal circumstances (because there is a good control over max supply from the USD value of the token), there could be malicious tokens $T$ registered on COMMON trying to violate this assumption by minting a huge number of units.

3.2 Constants

- **MAXSUPPLY** – upper bound on the maximum supply each token in COMMON can have.
- **$M$** – precision constant of FixedPoint. See Section 2.2.1 for more information on the FixedPoint type.
- **$B$** – upper bound on the numerator in FixedPoint,
- **$N$** – another precision constant, used to guarantee integral values under homomorphic encryption, see Section 2.2.4, we can assume $N|M$, in particular, $N$ is smaller than $M$.

\footnote{Note that the price of $T$ can then fluctuate in time, but it is safe to assume it does not go up or down by a factor of $10^6$.}
• LENCOLLECTPHASE – the number of blocks the collect phase is supposed to take. (See ORDER-BOOK.FinalizeCollectPhase.)
• AUCTIONLENGTH – the number of blocks the Dutch auction in SWAP-ENGINE is supposed to take.
• PRICESLACK – slack that we apply to current prices to obtain acceptable prices in the current batch. Should be a small positive constant, like 0.005, or can be 0.
• DUMMY – a dummy element of type Scalar, that has unknown preimage under the chosen hash function. We use it to signify "empty".
• HEIGHT – the default height of Merkle trees that we use. Example value HEIGHT = 30.

We also note that it’s convenient to choose all the constants: MAXSUPPLY, M, B, N as either powers of 10 or powers of 2.

3.3 Common Types

Apart from self-explanatory types like Int or Boolean we make use of the following custom types:

• Amount – used to represent token amounts. A value y of type Amount belongs to the range $[0, \text{MAXSUPPLY}]$ and represents a token amount. See 2.2.2 for more details.
• FixedPoint – a fraction value with fixed precision. See 2.2.1.
• Price – same as FixedPoint (type alias), but is specifically meant to store the price of a token with respect to another (see 2.2.3). Intuitively, the price being $p \in \mathbb{Q}$ for a token pair $(A, B)$ represents the fact that $p \cdot x$ tokens $A$ are worth roughly the same as $x$ tokens $B$.
• Ratio – a value of type FixedPoint but in the interval $[0, 1]$. See 2.2.3.
• ScaledRatio – an integer value $0 \leq y \leq N$ that represents the rational $y/N$. This fraction belongs to the range $[0, 1]$. See 2.2.4.
• ScaledAmount – an integer value $0 \leq a \leq \frac{\text{MAXSUPPLY}}{N}$ that represents the number $y \cdot N$. See 2.2.4.
• Token – a type of variable representing a unique identifier of a token, e.g. the address of the contract the token corresponds to.
• Pair – a type of the form $(\text{Token}, \text{Token})$. We refer to the first element of a pair as pair.from and to the second element as pair.to. In the context of trading, if an order refers to a pair $= (A, B)$ : Pair then it corresponds to the user’s intent to buy tokens $B = \text{pair.to}$ in return for $A = \text{pair.from}$.
• Round – Int representing a round number in COMMON,
• Phase – a type that describes which phase of the round the contract is currently in. Its possible values are collect, reveal, and trade.
- **OrderId** – an alias for **Scalar**, used to hold unique identifiers of orders.
- **AHCipherText** – a type representing an encrypted value. The encryption scheme (see Section 2.1.5) is additively homomorphic, hence the "AH" prefix.
- **EncPKey** – a type representing the Decryption Oracle’s public key for encryption, see Section 9.

### 3.4 Composite Types

In the pseudocode, especially when referring to data types, we use a syntax similar to Rust. For instance the following types appear in several places throughout Sections 6 and 7:

- **Map \langle T, S \rangle**: represents a key-value mapping with keys of type \( T \) and values of type \( S \).
- **Set \langle T \rangle**: represents a set of values of type \( T \).
- **Option \langle T \rangle**: the optional type, it is either \( \text{None} \) or a \( \text{Some}(t) \) with \( t \) being a value of type \( T \).

Apart from these generic data structures, below we describe a few data types and structures specific to **COMMON**.

#### 3.4.1 Nullifier Set

An important concept that has been introduced in [MGGR13, HBHW22] and then used in virtually all protocols that are based on shielded pools is that of a nullifier set (e.g. see ZEXE [BCG+18] and Privacy Pools [BIN+23]). This is simply a set of **Scalar** values which is held on-chain (in a smart contract, in the case of **COMMON**) and is used to invalidate notes (or more generally records) that have been spent, without pointing at a particular note. In Section 4.1 we give a conceptual explanation of how nullifiers are used.

```rust
struct NullifierSet
    // The set of nullified elements
    nullified : Set (Scalar);

Method: NullifierSet.initialize

1. Initialize self.nullified to an empty set.
```

The **NullifierSet.nullify** operation takes an element (from the \( F \) field) and outputs a boolean value signifying whether the element was newly inserted.
Method: NullifierSet.nullify

Input: element: Scalar
Output: Boolean

1. If element ∈ self.nullified:
   Return False
2. Insert element into self.nullified
3. Return True

3.4.2 Merkle Trees

Merkle trees [Mer87] is a concept used all over blockchain systems. For privacy solutions it is typically used to store a set of note (or record) hashes in a way that allows to efficiently add new elements to it, and then prove inclusion of specific elements without pointing to the explicitly. We refer to Section 4.1 for a high-level description of shielded pools that gives an idea of the significance of Merkle trees.

In this section we briefly describe how Merkle trees are implemented for our particular application. We refer to ZCash [HBHW22] for more details.

The MerkleTree structure is simply a full binary tree of a particular height. It is initialized with DUMMY elements to signify that it is empty. Later on, one can insert new elements to the tree (but never remove elements), and the newly inserted elements land in subsequent leaves of the tree. In the below implementation we also hold historicalRoots, which is the set of all roots the tree ever had. This is kept for technical reasons related to the fact that the proof of inclusion in the tree references a specific root, and a proof submitted to the contract that references a historical root (and not necessarily the current one) must still be recognized as correct.

```
struct MerkleTree
   // Height of the tree
   height: Int;
   // Tree vertices have ids between 1 to 2^height − 1
   vertices: Map(Int, Scalar);
   // The set of all roots the tree ever had.
   historicalRoots: Set<Scalar>;
   // Number of leafs that are occupied so far.
   numLeafsUsed: Int;
   // The set of all occupied leaves.
   leafSet: Set<Scalar>;
```

We note that the below initialization method MerkleTree.initialize is naive and is presented here only for the sake of simplicity, otherwise it should not be used in a real system. Instead, one should initialize the tree lazily (i.e., create vertices at the moment when they are referenced for the first time).
Method: MerkleTree.initialize

Input: height : Int

1. Set self.height = height
2. Initialize self.vertices by placing DUMMY in all $2^{height-1}$ leaves at the indices in the range $[2^{height-1}, 2^{height-1} - 1]$ and computing the remaining vertices as hashes of children.
3. Initialize self.historicalRoots to an empty set
4. Initialize self.leafSet to an empty set
5. Set self.numLeafsUsed = 0

Adding a leaf is done by placing the new element in the first free leaf spot and then recalculating the hashes along the path from the leaf to root. Thus the complexity is $O(self.height)$.

Method: MerkleTree.addLeaf

Input: leaf : Scalar

1. Assert numLeafsUsed $< 2^{height-1}$  // $2^{height-1}$ is the tree capacity
2. Place leaf at a leaf number self.numLeafsUsed in the tree (this corresponds to position $2^{height-1} + self.numLeafsUsed$ in self.vertices). Recalculate the hashes at vertices on the path from leaf to the root.
4. Insert root into self.historicalRoots
5. Insert leaf into self.leafSet
6. Set self.numLeafsUsed = self.numLeafsUsed + 1
7. Return self.numLeafsUsed - 1

Method: MerkleTree.isHistoricalRoot

Input: root : Scalar
Output: Boolean

1. Return [root $\in$ self.historicalRoots]

Method: MerkleTree.isLeaf

Input: leaf : Scalar
Output: Boolean

1. Return [leaf $\in$ self.leafSet]

Below we omit the technical details of Merkle proof generation as it is standard – the proof is just $\approx self.height$ field elements.
Method: MerkleTree.generateProof

<table>
<thead>
<tr>
<th>Input</th>
<th>leafHash: Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>(Scalar, MerkleProof)</td>
</tr>
</tbody>
</table>

1. Assert leafHash is one of the leaves in the tree. Let its position be leafId.
2. Compute merkleProof – the Merkle proof of inclusion of the leaf at position leafId.
3. Return (self.root, merkleProof)

3.4.3 Notes

Below we define the Note structure. The ORDER-BOOK.tokenBag Merkle tree holds hashes of Note (see Section 6).

```plaintext
struct Note
  // type of tokens the note holds
tokenId: Token;
  // amount of tokens the note holds
value: Amount;
  // secret held by the owner of the note
trapdoor: Scalar;
  // secret used to invalidate the note when spent
nullifier: Scalar;
```

3.4.4 Orders

As explained in Section 4, each order in COMMON is held in two parts – one part is fully public (the Order struct) and the second part is hidden and kept only as a hash in ORDER-BOOK.orderBag (the OrderNote struct).


```plaintext
struct Order
    // the fraction of the order that has been filled already
    fillRatio : Ratio;
    // the pair of tokens the user wants to trade
    pair : Pair;
    // the maximum price the user wants to pay to buy token pair.to for token pair.from (see also definition of Price)
    maxPrice : Price;
    // whether the order has been cancelled
    isCancelled : Bool;
    // the number of batch in which the order was placed the last time in case the fillRatio has not yet been updated after the trade, or None if the order is up-to-date
    lastBatch : Option(Round);
    // the encryption of the order amount
    encAmount : AHCipherText;
```

The types `Ratio` and `ScaledRatio` are separate, and in fact use different constants: M and N to define precision. That is why we need the below method to convert one type into the other.

**Method: Order.maxBoundedFraction**

**Output: ScaledRatio**

1. Set `scaledOrderFraction` to be the largest positive integer k such that \( k/N \leq 1 - self.fillRatio \)
2. Return `scaledOrderFraction`

We note that there is a certain amount of duplication between `Order` and `OrderNote`. This is intended and necessary – there are operations such as `ORDER-BOOK.ClaimSwapped` where (for privacy reasons) we don’t want to refer to a specific order, only prove the order exists and refer some of its data privately.

```plaintext
struct OrderNote
    orderId : OrderId;
    pair : Pair;
    scaledAmount : ScaledAmount;
    orderTrapdoor : Scalar;
    orderNullifier : Scalar;
```

### 3.4.5 Events

There are two types of events that we record in `ORDER-BOOK.eventLog`. 
struct OrderInBatchEvent
    // the order the event refers to
    orderId : OrderId;
    // the pair in the order
    pair : Pair;
    // fraction of the order that was placed in a batch (scaled by N)
    scaledOrderFraction : ScaledRatio;
    // the round the event took place
    round : Round;

struct TradeEvent
    // the pair that was traded
    pair : Pair;
    // fraction of the intended amount that was traded
    tradedFraction : Ratio;
    // the round then the event took place
    round : Round;
    // the price of the trade
    price : Price;

The general Event type is the union of the two types OrderInBatchEvent and TradeEvent.

union Event
    OrderInBatchEvent;
    TradeEvent;

Since in the protocol we need to compute $\text{Hash}(\text{event})$ for event : Event we should define what is meant by that. The simplest way to do that is to map elements of type Event into $\mathbb{F}^5$. Since each of the 4 fields in both types OrderInBatchEvent and TradeEvent map straightforwardly into $\mathbb{F}$, we can just map elements of OrderInBatchEvent into tuples of the form $(1, \cdot, \cdot, \cdot, \cdot)$ and elements of TradeEvent into tuples $(2, \cdot, \cdot, \cdot, \cdot)$. This mapping is obviously one-to-one, and as explained in Section 2.1.4 hashing tuples of $\mathbb{F}$ is exactly what $\text{Hash}()$ supports.

4 Technical Overview

4.1 Shielded Token Pool

The foundational component upon which COMMON is built is the Shielded Token Pool. A user holding regular tokens (ERC20 in case of EVM chains or PSP22 in case of Substrate,
see also Section 3.1.4) can deposit such tokens to the shielded pool (see Figure 13) and then withdraw them at any time (see Figure 14). The fundamental property of the shielded pool is that a third party observer is not able to link two different actions in the pool as coming from the same user. Thus, with the number of the shielded pool users growing, the anonymity set grows, and hence more privacy is gained. The idea for shielded pools has been first introduced in [MGGR13, HBHW22] and several variants have been studied subsequently [BCG18, PSS19, Wil18, EKKV22, Lab23]. All these solutions share a common core: a merkle tree with commitments to "notes" and a "nullifier set". They differ mostly in what data is exactly held in these notes and whether the system allows transfers within the shielded pool or only deposits and withdrawals. Below we give some details on how is a shielded pool implemented in COMMON.

Notes. The main component of a shielded pool is a Merkle tree ORDER-BOOK.tokenBag (see Section 3.4.2 for details) holding hashes of notes in its leaves (for a detailed definition of Note we refer to Section 3.4.3). The Merkle tree structure allows for efficient inclusion, and for efficient proofs that a given note hash is contained in the tree. Each note consists of:

- **tokenId** – think ETH, USDT or AZERO,
- **amount** – number of token units,
- **trapdoor** and **nullifier** – secret field elements that are required to track ownership and the spending rights of this note.

Depositing tokens. Suppose a user holds 10 AZERO tokens and would like to deposit them to the shielded pool. The precise steps such a user should perform are described in Figure 15. Let us briefly review these steps here. The main step is for the user to submit an ORDER-BOOK.DepositTokens transaction containing:

- **noteHash** = Hash(note) (see Section 3.4.3 for details on notes),
- **proof**.

The above note satisfies note.tokenId = AZERO and note.amount = 10. Moreover, the note has note.nullifier and note.trapdoor generated uniformly at random from F by the user. The user is expected to store these and keep them secret. The proof attached to the transaction is a zk-SNARK showing that noteHash is indeed the hash of a note that has a correct tokenId and amount – see the relation R_DepositTokens in Section 8.9.

The ORDER-BOOK contract after receiving such a transaction, verifies the proof, and adds noteHash as a new leaf in the merkle tree. The user is the only holder of nullifier and trapdoor and hence has exclusive rights to spend the note: either by withdrawing, or performing another action, such as creating a buy order.

Spending notes. Above we have explained how a user can deposit tokens in the shielded pool (implemented as part of the ORDER-BOOK contract) in order to add its note to the

---

8Technically note.amount would be 10 \cdot 10^{12} because we keep token amount integral, and in this example 12 is the number of decimals of AZERO.
ORDER-BOOK.tokenBag Merkle tree. It is instructive to observe that at this point every user of the blockchain is aware that a very particular leaf in the tree is a hash of a note containing 10 AZERO. So to preserve anonymity it is crucial to not refer explicitly to a particular leaf of the tree when spending the note. Let us consider the simplest case of spending: withdrawing the token. To do so, the user needs to call the ORDER-BOOK.WithdrawTokens method of the smart contract supplying suitable data. Apart from the tokenId = AZERO and the amount = 10, the user needs to attach proof and nullifier. The proof is a zk-SNARK that there exists a noteHash in the ORDER-BOOK.tokenBag Merkle tree – such a proof does not point to a particular leaf in the tree, the leaf (as well as a corresponding merkle branch) is part of the witness data, thus is not revealed. However, to prevent the user from spending a note multiple times, the nullifier must be revealed. The ORDER-BOOK contract checks that the revealed nullifier has not been used before, and then stores it in the tokenNullifierSet (exactly to prevent spending the note again).

Merging and splitting notes. Once a note lands in the shielded pool, it’s value cannot be altered. This is quite limiting, hence the shielded pool also allows to arbitrarily split one note into multiple smaller notes, or combine multiple notes to yield one (cf. Section [6.2.7]). Technically, merging is just withdrawing two (or more) notes and depositing one being a sum – no new technical idea is required to achieve that, other than what we have discussed above for deposit and withdraw. However, the merge and split operations allow to not reveal the individual values of notes, hence they cannot be directly "emulated" with these operations.

Using shielded pools. It is worth noting that shielded pools are effective in providing privacy only if the users hold the funds inside the pool long enough. Indeed, depositing the tokens to the pool and withdrawing them shortly after (especially if it’s the same amount, see also Section [10.4]) is not recommended, since the time correlation of these two events might put a high probability link between them. Instead, the ideal way of using shielded pools is to hold all the tokens there, thus essentially never withdrawing from the pool, except when this is really necessary. One of the main reasons this is not particularly popular for legacy solutions, is that interactions with DeFi protocols or any other contracts is not supported directly from shielded pools (one must withdraw into the clear). COMMON allows for trading directly from the shielded pools, thus improving the user experience in this aspect – with more DeFi being natively integrated with shielded pools holding tokens in the pool by default might become a convenient option.

4.2 COMMON – a Bird’s Eye View

Technically, COMMON consists of two interoperating smart contracts: the ORDER-BOOK and the SWAP-ENGINE and one additional off-chain component: the Decryption Oracle. To best describe what the responsibilities of each of those components are, we analyze the flow of interacting with COMMON from the perspective of a user.

First of all – the only way of trading on COMMON is by sending limit orders of the form "I want to buy tokens Y for some amount of tokens X at price at most p". Note that apart from classical long-living orders in the order book (waiting to be matched)
this also captures spot trading: one can just specify \( p = \infty \) in order for the trade to happen at the current price. Thus, even though the interface is simplistic it allows for both typical ways of trading. In the below description we focus on one trading pair only: (AZERO, ETH), and for simplicity we assume that there is only this single pair in the DEX.

**Creating Orders.** Suppose that we have 1000 AZERO tokens in the shielded pool and that we are willing to buy 1 ETH for them. To this end we should issue a transaction `ORDER-BOOK.CreateOrder` which can be seen as registering an intent of the form "I want to buy tokens ETH for some amount of tokens AZERO at price at most 1000.0 (see also the definition of `Price` and `Pair` in Section 3.3). Note that here, we intentionally skip how many of AZERO tokens the order is about – while the token pair (AZERO, ETH) is public in the order, the token amount remains private. Indeed, the `ORDER-BOOK.CreateOrder` along with cryptographic proofs that the order is well formed contains an encrypted amount (in this case 1000 AZERO) that only a special entity – the Decryption Oracle can decrypt. The Decryption Oracle is a black-box abstraction of a component that will decrypt only specific ciphertexts, and only when certain conditions are satisfied on-chain. Concretely, one can instantiate the Decryption Oracle using Multi-Party Computation – we refer to Section 9 for details – we also give there alternative instantiations: using TEEs or just a single trusted party.

**Rounds and Batches.** After such an order is placed, it waits for being filled. COMMON operates in rounds, each round has a prespecified length. Each round consists, roughly speaking, of: 1) selecting orders that we want to fill this round, 2) trying to fill these orders. Our example order for the pair (AZERO, ETH) will stay dormant in rounds when the spot AZERO/ETH price stays above 1000, but after it drops below to, say 999.9, the order will be selected for the round’s batch of orders. For the (AZERO, ETH) pair, each buy order above price 999.9 will be selected to the batch, as well as each sell order below price 999.9. After the collect phase of the round is over, all the orders in both directions are aggregated, to form "batched orders": one in the direction (AZERO, ETH), and another in the opposite direction (ETH, AZERO). Note that each order in such a batch is encrypted (has unknown) value, and what the batching essentially does (details in Subsection 4.3) is homomorphic addition of all these encrypted orders so that they become one large order (for each direction). After that, the Decryption Oracle is queried to decrypt the aggregated orders, and the result triggers the next phase: trade.

**Trading.** The trade phase is realized by the `SWAP-ENGINE`. It is worth clarifying that this round does not involve privacy, and everything happens in the clear. Roughly speaking, the `SWAP-ENGINE` receives the values of orders to be traded for all the pairs, along with the underlying tokens (the corresponding ERC20/PSP22 tokens are transferred from `ORDER-BOOK` to `SWAP-ENGINE`) and its goal is to fill the received orders as much as possible. The `SWAP-ENGINE` fills the order in two phases: 1) it matches internally between orders in

---

10Since we distinguish the pairs (AZERO, ETH) and (ETH, AZERO), formally we have 2 pairs. The first one corresponds to buying ETH for AZERO, and the second one, the opposite.

11Time can be measured using regular timestamps, yet the description in this paper uses block numbers.
opposite directions, 2) the rest is traded in an open auction that happens on a distance of $\text{AUCTIONLENGTH}$ blocks. More details are provided in Section 4.4, but the main idea is to let market makers bring liquidity from all possible markets in order to obtain best possible prices. Once the auction is over, the SWAP-ENGINE reports back to the ORDER-BOOK informing it about the amounts it managed to trade along with the resulting prices. ORDER-BOOK processes all this information and saves necessary data in the contract that is then used by users to claim the traded funds. An important note is in order here: the swap engine does not have the guarantee to fill the input orders in full. Indeed, if some order is huge, then there might not be enough liquidity to trade all of it in a single round (note that the SWAP-ENGINE must respect the limit prices requested by users). In such a case, the order will land in two or more subsequent batches, until it is filled fully.

Let us then assume that our example order was matched at 50% and the price was 999.8 (it is guaranteed to be at most 1000 – the price on the limit order).

**Claiming.** After an order is traded in a particular round (either fully or partially) the user is able to claim the swapped funds and bring them to the shielded pool. To this we need to send a ORDER-BOOK.ClaimSwapped transaction, referencing publicly the traded pair: (AZERO, ETH), the round when the trade happened, but keeping all the remaining details: the claimed value, and the particular orderId private. This transaction allows us to create a note containing $\frac{0.5 \cdot 1000}{999.8} \approx 0.5001$ ETH. The remaining 50% of the order, i.e., 500 AZERO is still pending, and will likely be filled in the subsequent round.

When it comes to claiming, there are no limitations as to when it is done, the user can wait arbitrarily long before claiming swapped tokens. Apart from that, it’s also possible to cancel an order and claim the tokens that were not traded.

In the subsequent subsections 4.3 and 4.4 we dive deeper into the internal workings of the ORDER-BOOK and the SWAP-ENGINE.

### 4.3 Order Book

**Order Storage.** Apart from the data associated with the shielded pool itself, the ORDER-BOOK holds also a number of other items, including: ordersSet – a map holding public data about specific orders, orderBag – a Merkle tree holding hashes of private data about the existing orders (similarly as tokenBag has hashes of private tokens), eventLog – a Merkle tree holding hashes of events happening during rounds. Whenever a user creates a new order by issuing the ORDER-BOOK.CreateOrder transaction, data is added to two places: ordersSet and orderBag. More specifically, the user creates two structures: an order (of type Order) and an orderNote (of type OrderNote), see Subsection 3.4.4 for details of what these comprise of. Roughly speaking, the order contains the public data about the order that will be stored in the contract in the plain (in ORDER-BOOK.ordersSet) and Hash(orderNote) will be added to ORDER-BOOK.orderBag, without revealing orderNote. The public order contains information such as order.pair$^{12}$

$^{12}$The ordered pair of tokens, for instance (AZERO, ETH) – means the order is about buying ETH and selling AZERO.
**order.maxPrice** – the limit price of buying, **order.encAmount** the encrypted amount of tokens to sell, and other data related to managing the order. The private **orderNote** on the other hand contains, importantly the plaintext **orderNote.amount** (note that this is secret since only Hash(orderNote) is stored on chain), and similarly to the token notes: **orderNote.orderNullifier** and **orderNote.orderTrapdoor** that serve similar purpose as the analogous for token notes. To guarantee consistency between these two pieces of data in the **ORDER-BOOK** the user submits a zero-knowledge proof (see Subsection 8.4) that the **orderNote** agrees with **order**.

**Order Filling.** Each order, when initialized as a result of the **ORDER-BOOK.CreateOrder** transaction starts with **order.fillRatio = 0**. The **fillRatio** determines what fraction of the order has been filled already, and is a public value (in contrast to the token **amount** of the order). Once the **fillRatio** becomes 1, the order is fully filled, and will not be included in any more batches. After an order is included in a batch (using the **ORDER-BOOK.PlaceOrderInBatch** transaction) in some round, the **fillRatio** is updated accordingly after the round is over. This update must be triggered via a transaction **ORDER-BOOK.UpdateOrder** either by the user itself or by a party called an Updater (see Subsection 6.4). Whenever an order with some **fillRatio** is included in a batch, it is placed there with an indication what fraction of the order is being traded\footnote{For technical reasons \textit{scaledOrderFraction} and \textit{fillRatio} have different types, and are held with different precision (we emphasize this by the "scaled" prefix in the name, see Subsection 3.3. Thus the equality \textit{scaledOrderFraction} = 1 – \textit{fillRatio} might hold only approximately. See also \textit{order.maxBoundedFraction}.} – \textit{scaledOrderFraction} = 1 – \textit{fillRatio}. The **scaledOrderFraction** value is used specifically in two places:

- when computing the encrypted amount in a partial order – see Figure 7
- when claiming swapped tokens – see Figure 4

**Batching Orders.** Whenever a new round starts in the **ORDER-BOOK**, the **collect** phase is initialized and for each **pair** (such as **pair** = (AZERO, ETH)) the **encAggregate[pair]** storage item in **ORDER-BOOK** is set to an encryption of 0. The idea is that **encAggregate[pair]** is the encrypted sum of all orders added to the current batch, for a specific **pair**. Adding new orders to the batch is done via the **ORDER-BOOK.PlaceOrderInBatch** transaction (see Figure 7), it is expected to be triggered by updaters, see Subsection 6.4). Here are a few necessary conditions that must be satisfied to add an order for a particular **pair** to the current batch:

- the order’s **fillRatio ≠ 1** and the order has not been cancelled,
- the order has been updated (using **ORDER-BOOK.UpdateOrder**) since the last time it was included in a batch (an indication for this is that **order.lastBatch = None**),
- the current rounds price for this pair: \( \text{maxBuyPrices[pair]} \leq \text{order.maxPrice} \).

The last condition is required, because once the orders are batched, the **SWAP-ENGINE** will be asked to trade the aggregate order at a price no worse than **maxBuyPrices**, so
including only orders with limits larger than that guarantees the user’s limit order price is satisfied. For each pair \( \text{maxBuyPrices}[\text{pair}] \) is determined using prices queried from a price oracle, adjusted by a small constant\(^{14}\) called \( \text{PRICESLACK} \), see Figure 9 how it is computed. Once an order is placed in a batch, the status of this order is updated by setting \( \text{order.lastBatch} = \text{currentRound} \) and the encrypted value is updated as

\[
\text{encAggregate}[\text{pair}] = \text{encAggregate}[\text{pair}] + \text{scaledOrderFraction} \cdot \text{order.\text{encAmount}}.
\]

Note that in the above, the values \( \text{encAggregate}[\text{pair}] \) and \( \text{order.\text{encAmount}} \) are ciphertexts, and \( \text{scaledOrderFraction} \) is a small integer\(^{15}\). However since the encryption scheme we use is additively homomorphic multiplication of a ciphertext by a scalar is possible to achieve (by the "repeated squaring" algorithm).

**Claiming Swapped Tokens.** An important element of ORDER-BOOK is the \( \text{eventLog} \) – it’s where the ORDER-BOOK deposits events\(^{16}\) more specifically:

- Upon placing an order in a batch, a \( \text{OrderInBatchEvent} \) is added to \( \text{eventLog} \), which contains information about the current round and the \( \text{orderId} \) of the order, see Figure 7
- Upon terminating the trade phase in the ORDER-BOOK, the \( \text{TradeEvent} \) is added to \( \text{eventLog} \) which for each pair traded in this round saves data on what fraction of the order was traded, and what was the average price, see Figure 12.

The events are necessary for the users to claim swapped tokens. More specifically, in order to claim the 0.5001 ETH that we have received as 50% of our order, we need to send the ORDER-BOOK.CClaimSwapped transaction. As part of the transaction a specific \( \text{swapOrderNullifier} \) is included and a proof that our claim is justified. We refer the reader to Subsection 8.8 to learn what is this proof about, but just to give some intuition: the proof must reference two events in the \( \text{eventLog} \), one \( \text{OrderInBatchEvent} \) and one \( \text{TradeEvent} \) (thus the proof checks a merkle proof of inclusion of these events) and connect these to a particular \( \text{orderNote} \). The \( \text{swapOrderNullifier} \) for this claim is computed as

\[
\text{swapOrderNullifier} = \text{Hash(\text{orderNote.orderNullifier}, \text{round})}
\]

(see Figure 19) – this way we can have multiple different nullifiers per the same order.

**Cancelling an Order.** To cancel an order the user is supposed to send the transaction ORDER-BOOK.CancelOrder. What this essentially does, is sets the \( \text{isCancelled} \) field of the order to True so that the order cannot be placed in any batch anymore. Afterwards, assuming the order is up-to-date already (this might require waiting till the

---

\(^{14}\)While in the paper \( \text{PRICESLACK} \) is a constant, independent on the \( \text{pair} \), a version where \( \text{PRICESLACK} \) is variable, depending on \( \text{pair} \) but also on what happened in previous rounds is certainly a viable variant of the protocol.

\(^{15}\)Even though \( \text{scaledOrderFraction} \) corresponds to a fraction in \([0, 1]\), we represent it as an integer, to enable homomorphic encryption. See the definitions of \( \text{ScaledRatio} \) and \( \text{ScaledAmount} \) in Section 3.3

\(^{16}\)More precisely, hashes of events are stored there.
end of the round), the user can claim the tokens that were left unswapped using the ORDER-BOOK.ClaimCancelled transaction. The content of this transaction is similar to ClaimSwapped, but referencing events is not necessary for the case of cancelling. Apart from that, the cancelOrderNullifier has the form

\[ \text{cancelOrderNullifier} = \text{Hash}(\text{orderNullifier}, \text{cancel}), \]

where cancel is simply a special field element guaranteed to be not equal to any round number.

### 4.4 Swap Engine

The role of SWAP-ENGINE is to perform the trades coming from the ORDER-BOOK. Specifically, after the trade values are revealed, the ORDER-BOOK calls the SWAP-ENGINE.Start method in order to initialize the SWAP-ENGINE. There are two inputs to this function:

- **amountsFrom**: Map(Pair, Amount) a mapping representing for each pair \((A, B)\) how many tokens \(A\) is the SWAP-ENGINE supposed to trade for tokens \(B\),

- **maxBuyPrices**: Map(Pair, Price) a mapping determining for each pair \((A, B)\) the maximum possible price for buying tokens \(B\) for tokens \(A\).

The goal of the SWAP-ENGINE is, for each pair \((A, B)\), to trade as many tokens \(A\) for \(B\) as possible (but at most \(\text{amountsFrom}[(A, B)]\), while respecting the given bound on the price \(\text{maxBuyPrices}[(A, B)]\). Once the SWAP-ENGINE is done with its trades (for which it spends a prespecified amount of time \(\text{AUCTIONLENGTH}\)) it returns the result to the ORDER-BOOK by calling ORDER-BOOK.FinalizeTradePhase. The output consists of two items:

- **sold**: Map(Pair, Amount) – for each pair \((A, B)\) how many of the \(A\) tokens did the SWAP-ENGINE manage to swap for \(B\) tokens,

- **bought**: Map(Pair, Amount) – for each pair \((A, B)\) how many of the \(B\) tokens did the SWAP-ENGINE managed to receive in return for the sold\([\!(A, B)\!]\) tokens \(A\).

Both when ORDER-BOOK is calling the SWAP-ENGINE and when SWAP-ENGINE is calling ORDER-BOOK back, the corresponding amounts of tokens are transferred between the contracts. Having explained how the SWAP-ENGINE interacts with the ORDER-BOOK we are ready to explain how the trading is done – it is performed in two phases, which we discuss separately below.

**Internal matching.** In order to maximize the amounts traded, as well as optimize the prices for users, the SWAP-ENGINE starts by matching the opposite pairs against each other. If for a pair of tokens \((A, B)\), both constraints \(\text{amountsFrom}[(A, B)] > 0\) and \(\text{amountsFrom}[(B, A)] > 0\) are satisfied, the SWAP-ENGINE first determines a common price \(p\) such that:

\[ p \leq \text{maxBuyPrices}[(A, B)] \quad \text{and} \quad p^{-1} \leq \text{maxBuyPrices}[(B, A)], \]
and then trades the maximum amount of $A$ and $B$ tokens so as to not exceed any of $\text{amountsFrom}[(A, B)]$, $\text{amountsFrom}[(B, A)]$. Because of how $\text{ORDER-BOOK}$ determines $\text{maxBuyPrices}$ it is guaranteed that such a price $p$ exists and can be, for instance the geometric mean of $\text{maxBuyPrices}[(A, B)]$ and $\text{maxBuyPrices}[(B, A)]^{-1}$. We refer to Figure 23 for the details.

In this version of the protocol the internal matching "closes" cycles of length 2 only: $A \rightarrow B \rightarrow A$. One can generalize this to a version where more sophisticated internal trades are made in order to maximize the traded amounts, this generalization is left out of the scope of this paper. It is worth mentioning that matching internally is heavily preferred over selling in the second phase, since there is no trading fee involved in the first phase, which allows to secure better prices.

It is expected that the $\text{SWAP-ENGINE}$ does not manage to fill all the trading requests it got from the $\text{ORDER-BOOK}$ during the internal matching phase. In the second phase, the $\text{SWAP-ENGINE}$ runs for each pair $(A, B)$ (independently, in parallel) a Dutch auction to buy $B$ tokens for $A$ tokens. The Dutch auction proceeds in $\text{AUCTIONLENGTH}$ blocks. It starts by computing an initial price $p_0$ that is suitably lower than $\text{maxBuyPrices}[(A, B)]$ and it allows the market makers to sell tokens $B$ for $A$ at this initial price $p_0$. With every block, the price is increased, linearly, so that in the last block, the price is exactly $\text{maxBuyPrices}[(A, B)]$. We refer to Figure 21 for the implementation of the Dutch Auction mechanism – it is worth mentioning that the price at a given block is computable directly from the data the $\text{SWAP-ENGINE}$ contract holds and the block number, hence the price does not need to be constantly updated in the contract.

The mechanism used by the $\text{SWAP-ENGINE}$ allows to effectively aggregate all on-chain sources of liquidity. Indeed, if there is another DEX that allows to buy tokens $B$ for $A$ at a price cheaper than the current price of the Dutch auction, the market makers will naturally take advantage of such an arbitrage opportunity and buy tokens $B$ on the DEX to sell in the $\text{COMMON}$ Dutch auction.

We note that using an auction with a gradually changing price instead of just setting the price to $\text{maxBuyPrices}[(A, B)]$ right away allows $\text{COMMON}$ to secure better prices from the on-chain liquidity, as otherwise the cut of the arbitragers would be potentially higher. Moreover, we claim that in the case of auction the competition between market makers is expected to be healthier. Indeed, instead of MEV bots fighting for their transactions to take the whole trade (in case it is profitable), the auction favors market makers who can outbid the others by offering the best price. Since the auction takes several blocks of time, it is also possible for market makers to pull liquidity from different sources to perform non-atomic arbitrage (by bridging from different chains, or by bringing capital from centralized exchanges).

## 5 Security and Privacy

We provide an informal discussion of the guarantees provided by $\text{COMMON}$ in terms of security and privacy.
5.1 Security Guarantees for Users

Assumptions. On top of the blockchain assumptions stated in Subsection 3.1, we make the following cryptographic assumptions necessary to reason about the security of COMMON.

2. Completeness, Soundness and the Zero Knowledge property (ZK) of the underlying zk-SNARK.
4. The Decryption Oracle instantiation securely realizes the functionality described in Section 9.

Discussion. We consider the scenario when an honest user tries to exchange an amount of \(x\) token A for token B with maxPrice \(p\) on COMMON. The expected result is that the user is able to collect:

- \(\frac{y}{p}\) token B,
- and \(x - y\) token A,

where \(y\) is an amount that depends on how much time (number of batches) the user is willing to wait before it cancels its order and collects the untraded amount \((x - y)\) – note that the market conditions also play a crucial role here. This is the expected result because of the following reasons.

- **Impossibility for a malicious user to collect more than it should via submitting incorrect orders or claims.** The honest users are protected from other malicious users who trade on COMMON. In more detail, if a malicious user creates an order to exchange \(x\) token A for token B by spending a note (whose hash is included in the tokenBag) that proves ownership of \(x\) token A, then it will not be able to collect (i) more token A than the untraded amount of its order and (ii) more token B than the amount which corresponds to the fraction of its order that has been already traded and the prices of the corresponding batches in which this order was included. This is guaranteed by the soundness property of the underlying zk-SNARK, the relations used in the zk-SNARK (cf. Section 8), and the collision resistance property of the hash function used in the Merkle trees that we utilize for membership proofs (tokenBag, orderBag, eventLog). At a high level, every user proves (via the relation \(R_{newOrder}\)) in a zk-SNARK that it constructed correctly both the encryption of the order’s value and the order note (whose hash will be stored in the orderBag) using the same amount as the value of the note that it spends (whose hash is stored in the tokenBag).
In addition, when a user collects back the amount of its order that has not yet been traded or the traded amount by creating a note (whose hash will be included in the tokenBag), it proves among others in a zk-SNARK (via the relations $R_{\text{ClaimCancelled}}$ and $R_{\text{ClaimSwapped}}$ respectively) that the note is correctly constructed and includes the correct amounts and token IDs.

- **Impossibility for a malicious user to double-claim.** $\text{COMMON}$ prevents a user from spending twice a note from tokenBag via the use of nullifiers (as in [HBHW22]) whose correct use is checked in the relevant relations. Also, $\text{COMMON}$ prevents users from claiming twice the untraded or the traded amount of their orders by utilizing again nullifiers but in a more complicated way, as the same order can be executed in different batches. For example, recall that the nullifier the user makes public to collect the traded amount of its order that corresponds to a specific batch is constructed by hashing an initial nullifier that was stored in the order note concatenated with the round when this batch was created.

- **The order/claim/cancellation of an honest user will be accepted.** Due to the completeness property of the underlying zk-SNARK an honest user is able to construct zk-SNARK proofs that pass the verification tests.

- **An attacker cannot steal the funds from honest users due to privacy leakage.** Due to the ZK property of the underlying zk-SNARK, the preimage resistance property of the underlying hash function and the security guarantees of the Encryption Scheme used in the instantiation of the Decryption Oracle, an honest user does not reveal secret information about its notes and its order notes that could enable an attacker to spend them on its behalf.

- **The aggregated amounts of every batch that are sent to the SWAP-ENGINE are equal to the sum of the order values in the batch.** Assuming (i) that the Decryption Oracle instantiation securely realizes the functionality described in Section 9 and (ii) the Additive Homomorphism of the Encryption Scheme used in the instantiation of the Decryption Oracle, then the decrypted amounts that are submitted by the Decryption Oracle to the ORDER-BOOK (and later are sent to the SWAP-ENGINE) correspond to the sum of the order values.

- **Everyone can check the correct execution of $\text{COMMON}$.** All the procedures of $\text{COMMON}$ except the decryption of the aggregated amounts (its correctness can still be verified on chain) are performed on chain via smart contracts which means that everyone can check the correctness of the computations (cf. Subsection 3.1).

- **The Market Makers cannot steal funds from the users.** This holds because when the Market Makers submit a transaction in order to participate in the Dutch auction, then the smart contract SWAP-ENGINE checks that the Market Maker has given access to the correct amount of token that it wants to sell.

- **No fraction of the user’s order can be traded at a worst price than the upper bound $\maxPrice$ that it has set when it submitted its order.** We explain why this holds in a different paragraph later.
5.2 Security Guarantees for Market Makers

When a Market Maker sends a transaction to participate in the Dutch auction, then it specifies (put as input) the worst price (the lowest price for selling) that it is willing to accept. If this transaction gets included in the chain later than the current round (e.g. in the trade phase of the next round where a different price for the Dutch auction has been set), it will not be processed if the new price is worse than what the Market Maker had set as input. Also, if this transaction gets included in the collect or reveal phase of a next round then it will not be processed at all as there will be zero amount for trading available. Furthermore, a Market Maker that wants to exchange token B for token A needs to set as input in its transaction the maximum amount \( y \) of token A that it wants to buy and also to “send” the correct amount of token B according to the Dutch auction price. If there are not \( y \) token A available when its transaction is included (because for example other Market makers’ transactions were included first) then it will buy as much as possible and the remaining amount of the token B that the Market Maker sent but was not traded is returned. As a result, even if Market Maker’s transaction is included with a delay, a Market Maker does not lose the amount that has sent but was not traded, and also the traded amount is always executed at a price that is no worse than what it has specified.

5.3 Privacy Guarantees

Under the following assumptions, an honest user keeps secret (i) the note from the shielded pool that it spends in order to create a new order and trade on COMMON or the order note it consumes when it claims its funds after trading on COMMON and transfers them to the shielded pool, and (ii) what is the value of the newly created order. When the user trades on COMMON, it reveals the token IDs that it wants to sell and buy via its order (the direction of the order) and the maximum buying price that it is willing to accept. Note that although in our protocol a user reveals the direction, this can be mitigated by generating fake orders that encrypt “0” – see Subsection 10.5.

1. Preimage resistance property of the underlying hash function. We need this assumption in order to retain secret the notes and the order notes of the users when they reveal the hash of them (recall that the hash values are included in the tokenBag and orderBag respectively).

2. ZK property of the underlying zk-SNARK. This property is essential so that the users do not reveal any secret information when they prove via a zk-SNARK that they own a note in tokenBag or an order note in orderBag with specific characteristics.


4. We assume that the adversary cannot corrupt all but one parties whose orders will be included in the same batch. Note that this assumption is needed for every private
DEX that uses batching as the technique to hide the orders’ value (it makes public the aggregated amounts). This assumption is essential because the aggregated amounts of a pair and direction are revealed, so if the adversary corrupts all but one parties in the batch, it can learn the value of the remaining user in the batch.

5.4 Price

Let us first discuss which is the worst case price. Every order in the same batch is traded at the same price \( p_0 \) which does not exceed the \( \text{maxPrice} \) of any order that is included in the batch. Note that before the end of every round, the ORDER-BOOK computes which is the fraction of the batch that has been traded and which is the clearing price (for every pair and direction). This price depends on the price at which the internal matching performed and the price at which the Market Makers participated in the Dutch auction (both handled by the SWAP-ENGINE). For example, if after a round \( x\% \) of the batch has been traded at price \( p_0 \), then every user can claim the tokens that correspond to the trade of \( x\% \) part of its order and to price \( p_0 \).

This clearing price \( p_0 \) is no worse than the \( \text{maxPrice} \) of any order in the batch, because:

1. The ORDER-BOOK collects for every batch only orders whose \( \text{maxPrice} \) is higher than the upper bound on the buying prices that it can guarantee for this batch (this is denoted by \( \text{maxBuyPrices} \) and is equal to an oracle price \( \text{currentPrices} \) multiplied by some tolerance \( (1 + \text{PRICESLACK}) \), where \( \text{PRICESLACK} \) is a parameter).

2. The SWAP-ENGINE that is responsible for the trading of the aggregated amounts (i) performs an internal matching that respects the maximum prices of all the directed pairs. In more detail, it achieves this by selecting a price for the directed pair \((A, B)\) that is between \( \text{maxBuyPrices}[(B, A)]^{-1} \) and \( \text{maxBuyPrices}[(A, B)] \); if no such price exists then it skips the internal matching. (ii) queries again the price oracle after the internal matching, but the Dutch auction starts with the oracle prices only if they respect \( \text{maxBuyPrices} \); otherwise \( \text{maxBuyPrices} \) are used for the initial Dutch auction price) (iii) as the Dutch auction progresses, although the prices become worse for the users and better for the Market Makers, they never become worse than \( \text{maxBuyPrices} \).

Best case price: if the oracle price queried by the SWAP-ENGINE for \((A,B)\) is smaller than \( \text{maxBuyPrices}[(A, B)] \), then in the best case the price for \((A, B)\) (price for buying) will be between the geometric mean of \( \text{maxBuyPrices}[(B, A)]^{-1} \) and \( \text{maxBuyPrices}[(A, B)] \), and this oracle price (we do not know which of the two is smaller). This holds because in the best case some fraction of the aggregated amount for pair \((A, B)\) will be traded via internal matching at a price that is equal to the geometric mean of \( \text{maxBuyPrices}[(B, A)]^{-1} \) and \( \text{maxBuyPrices}[(A, B)] \) and some other fraction will be bought by the Market Makers at the beginning of the Dutch auction where the price is equal to the oracle price. On the other side, if the oracle price is higher than \( \text{maxBuyPrices}[(A, B)] \), then the price will be between the geometric mean of the \( \text{maxBuyPrices}[(B, A)]^{-1} \) and \( \text{maxBuyPrices}[(A, B)] \) and \( \text{maxBuyPrices}[(A, B)] \).
For example, when \( \text{PRICESLACK} = 0 \) then \( \text{maxBuyPrices}[(A, B)] \) is equal to oracle price as queried by the \text{ORDER-BOOK} and thus equal to \( \text{maxBuyPrices}[(B, A)]^{-1} \). In that case the price for \((A, B)\) would be in the best case between oracle price as queried by the \text{ORDER-BOOK} and the minimum between the two oracle prices (the one that was queried initially by the \text{ORDER-BOOK} and the other queried by the \text{SWAP-ENGINE}) and in the worst case equal to oracle price as queried by the \text{ORDER-BOOK}. Note that the smaller the \text{PRICESLACK} is the better the price for the users but the more difficult to attract Market Makers.

6 Order Book

This section is devoted to describing the details of the \text{ORDER-BOOK} contract. For a high level description we refer to Subsection 4.3. The first part of the description is the definition of the \text{ORDER-BOOK} storage in Subsection 6.1. Subsequently in Subsection 6.2 all the contract calls of \text{ORDER-BOOK} are written as pseudocode. Finally in Subsection 6.3 we describe how would a user interact with the \text{ORDER-BOOK} contract: how should they craft transactions and which data should they preserve in their local storage.

6.1 Storage

We list all the storage items of \text{ORDER-BOOK} along with their types. In the pseudocode a given storage item, like \text{tokenBag}, is referred to as \text{self.tokenBag} because \text{self} is the \text{ORDER-BOOK} itself.

- \text{tokenBag} : \text{MerkleTree} – a Merkle Tree representing ownership of tokens by users. Each leaf of this tree is an element of type \text{Scalar} of the form \( h = \text{Hash} (\text{note}) \) where \text{note} is of type \text{Note}.

- \text{tokenNullifierSet} : \text{NullifierSet} – a set of field elements that represent invalidation of notes in \text{tokenBag}. More specifically, if \( n \in \text{tokenNullifierSet} \) then it is not possible to spend a \text{note} such that \text{note.nullifier} = n.

- \text{orderBag} : \text{MerkleTree} – holds secret data in users’ orders. Each leaf of this tree is an element of type \text{Scalar} of the form \( h = \text{Hash} (\text{orderNote}) \) where \text{orderNote} is of type \text{OrderNote}.

- \text{orderNullifierSet} : \text{NullifierSet} – a set holding invalidations of token claims resulting from swaps and cancelled orders. More specifically, each element of this set is of the following form: \( \text{Hash} (n, \text{nonce}) \) where:
  - \( n \) is a nullifier of some order in the \text{orderBag},
  - \text{nonce} is either an element of type \text{Round} (i.e. it is a round number), or the special element \text{cancel} : \text{Scalar} guaranteed to be not equal to any round number.

Each order can have multiple nullifiers in \text{orderNullifierSet} corresponding to claiming tokens swapped in various rounds and/or tokens claimed for cancelling the order.
• ordersSet : Map(OrderId, Order) – a mapping from order identifiers to the public order data.
• eventLog : MerkleTree – holds events that occurred in the order book. More specifically, the events are of type Event (see the definition of Event) and are held in the subsequent leaves of a Merkle tree as hashes.
• encAggregate : Map(Pair, AHCipherText) – a mapping representing the encrypted total trade value for each pair of tokens in the current round.
• aggregate : Map(Pair, Amount) – similar as encAggregate but the amounts are in the plain. Note that this map is populated only after encAggregate gets decrypted by the Decryption Oracle.
• maxBuyPrices : Map(Pair, Price) – a mapping representing the maximum price that the pair will trade at, the SWAP-ENGINE will generally try to buy as cheap as possible but it also has the hard constraint to never buy at a price higher than maxBuyPrices.
• encPK : EncPK – the public key of the Decryption Oracle used for encryption.
• currentRound : Round – the number of the current round.
• currentPhase : Phase – the current phase in round.
• currentRoundStart : Int – the block number when the current round has started.

6.2 Calls

This subsection is devoted to presenting the pseudocode of all the calls available in ORDER-BOOK. The calls often use zk-SNARK verification with respect to various relations \( R_\ast \) – the description of those can be found in Section 8.
6.2.1 Order Management

Public Call: ORDER-BOOK.NewOrder

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Assert orderId ( \notin ) self.ordersSet // ensure that the orderId is not taken</td>
</tr>
<tr>
<td>2. Assert self.tokenBag.isHistoricalRoot(root)</td>
</tr>
<tr>
<td>3. Assign ( x_{\text{NewOrder}} = (\text{root}, \text{noteHash}, \text{tokenNullifier}, \text{orderHash}, \text{encAmount}, \text{self}.\text{encPKey}) )</td>
</tr>
<tr>
<td>4. Assert ZKP.V(R_{\text{NewOrder}}, x_{\text{NewOrder}}, proof)</td>
</tr>
<tr>
<td>5. Assert self.tokenNullifierSet.nullify(tokenNullifier)</td>
</tr>
<tr>
<td>6. self.orderBag.Add(orderHash)</td>
</tr>
<tr>
<td>7. Initialize order: Order</td>
</tr>
<tr>
<td>• order.fillRatio = 0,</td>
</tr>
<tr>
<td>• order.pair = pair,</td>
</tr>
<tr>
<td>• order.maxPrice = maxPrice,</td>
</tr>
<tr>
<td>• order.isCancelled = False,</td>
</tr>
<tr>
<td>• order.lastBatch = None,</td>
</tr>
<tr>
<td>• order.encAmount = encAmount,</td>
</tr>
</tbody>
</table>

Figure 1: Creating a new order.

Public Call: ORDER-BOOK.CancelOrder

<table>
<thead>
<tr>
<th>Input: orderId: OrderId, root: Scalar, proof: ZKProof</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Assert self.orderBag.isHistoricalRoot(root)</td>
</tr>
<tr>
<td>2. Assign ( x_{\text{CancelOrder}} = (\text{root}, \text{orderId}) )</td>
</tr>
<tr>
<td>3. Assert ZKP.V(R_{\text{CancelOrder}}, x_{\text{CancelOrder}}, proof)</td>
</tr>
<tr>
<td>4. Set self.ordersSet[orderId].isCancelled = True</td>
</tr>
</tbody>
</table>

Figure 2: Cancelling an order.
6.2.2 Claiming Tokens

**Public Call: ORDER-BOOK.ClaimCancelled**

**Input:**
- cancelOrderNullifier: Scalar
- rootOrderBag: Scalar
- orderId: OrderId
- noteHash: Scalar
- proof: ZKProof

1. Assert self.orderNullifierSet.nullify(cancelOrderNullifier)
2. Set order = self.ordersSet[orderId]
3. Assert self.orderBag.isHistoricalRoot(rootOrderBag)
4. Assert (order.isCancelled = True) ∧ (order.lastBatch = None)
   // We ensure that the order has been marked as cancelled and is up-to-date
5. Assign x = (cancelOrderNullifier, rootOrderBag, orderId, noteHash, order.fillRatio)
6. Assert ZKP.V(R_ClaimCancelled, x, proof)
7. self.tokenBag.Add(noteHash)

Figure 3: Claiming funds from a cancelled.

**Public Call: ORDER-BOOK.ClaimSwapped**

**Input:**
- swapOrderNullifier: Scalar
- rootOrderBag: Scalar
- rootEventLog: Scalar
- noteHash: Scalar
- proof: ZKProof

1. Assert self.orderNullifierSet.nullify(swapOrderNullifier)
2. Assert self.orderBag.isHistoricalRoot(rootOrderBag)
3. Assert self.eventLog.isHistoricalRoot(rootEventLog)
4. Assign x = (swapOrderNullifier, rootOrderBag, rootEventLog, noteHash)
5. Assert ZKP.V(R_ClaimSwapped, x, proof)
6. self.tokenBag.Add(noteHash)

Figure 4: Claiming funds from a swap.

6.2.3 Adding Events

Note that the below call is internal – only the ORDER-BOOK can deposit events.

**Internal Call: ORDER-BOOK.AddEvent**

**Input:**
- event: Event

1. Set eventHash = Hash(event) // we refer to Section 3.4.3 for hashing Event.
2. self.eventLog.addLeaf(eventHash)

Figure 5: Adding an event to the log.
6.2.4 Batch Management

We recall that + and \cdot below are the arithmetic operations we defined for FixedPoint, Amount and Ratio data types (cf. Section 2.2.1).

Public Call: ORDER-BOOK.UpdateOrder

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td>orderId: OrderId, event: TradeEvent</td>
</tr>
<tr>
<td></td>
<td>1. Assert self.eventLog.isLeaf(Hash(event)).</td>
</tr>
<tr>
<td></td>
<td>2. Set order = self.ordersSet[orderId]</td>
</tr>
<tr>
<td></td>
<td>3. Assert order.lastBatch = event.round</td>
</tr>
<tr>
<td></td>
<td>4. Set scaledOrderFraction = order.maxBoundedFraction()</td>
</tr>
<tr>
<td></td>
<td>// it is the same fraction as was included in the batch in round event.round.</td>
</tr>
<tr>
<td></td>
<td>5. Convert scaledOrderFraction from type ScaledRatio to FixedPoint</td>
</tr>
<tr>
<td></td>
<td>6. Set order.fillRatio = order.fillRatio + scaledOrderFraction \cdot event.tradedFraction</td>
</tr>
<tr>
<td></td>
<td>7. Set order.lastBatch = None</td>
</tr>
<tr>
<td></td>
<td>8. Set self.ordersSet[orderId] = order</td>
</tr>
</tbody>
</table>

*a scaledOrderFraction is of type ScaledRatio thus really some \( y \in [0, N] \) representing \( \frac{y}{N} \). Since we assume \( N \) is a divisor of \( M \), this conversion to FixedPoint incurs no error.*

Figure 6: Update order.
### Public Call: ORDER-BOOK.PlaceOrderInBatch

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td>orderId : OrderId, round : Round</td>
</tr>
<tr>
<td>1.</td>
<td>Assert self.currentPhase = collect</td>
</tr>
<tr>
<td>2.</td>
<td>Assert self.currentRound = round</td>
</tr>
<tr>
<td>3.</td>
<td>Set order = self.ordersSet[orderId]</td>
</tr>
<tr>
<td>4.</td>
<td>Assert order.lastBatch = None</td>
</tr>
<tr>
<td>5.</td>
<td>Assert order.isCancelled = False</td>
</tr>
</tbody>
</table>
| 6. | Assert self.maxBuyPrices[pair] ≤ order.maxPrice  
    // this guarantees the price in the order will be respected |
| 7. | Set scaledOrderFraction = order.maxBoundedFraction() |
| 8. | Assert scaledOrderFraction > 0  
    // no point in trading 0 amounts |
| 9. | Initialize event: OrderInBatchEvent  
    - event.orderId = orderId  
    - event.pair = order.pair  
    - event.scaledOrderFraction = scaledOrderFraction  
    - event.round = round |
| 10. | self.AddEvent(event) |
| 11. | Set order.lastBatch = round |
    // homomorphic operations on ciphertexts |

Figure 7: Placing order in a batch.

#### 6.2.5 Control Flow

The call below initializes the contract and is assumed to be called just once, at the start, so it is really a constructor.
**Internal Call: ORDER-BOOK.InitializeOrderBook**

1. Call ORDER-BOOK.InitializeRound(1)
2. Set self.encParams = DecryptionOracle.KGen[0]
4. Initialize Merkle trees:
   - self.tokenBag.initialize(HEIGHT)
   - self.orderBag.initialize(HEIGHT)
   - self.eventLog.initialize(HEIGHT)
5. Initialize nullifier sets:
   - self.orderNullifierSet.initialize()
   - self.tokenNullifierSet.initialize()

Figure 8: Initialize contract.

**Internal Call: ORDER-BOOK.InitializeRound**

**Input:** round : Round

1. Set self.currentRound = round.
2. Set self.currentPhase = collect.
3. Set self.currentRoundStart = currentBlock
4. Call PRICE-ORACLE.QueryPrices() to populate currentPrices[pair] for each pair of tokens traded in ORDER-BOOK.
5. For each pair ∈ currentPrices:
   - Set self.maxBuyPrices[pair] = (1 + PRICESLACK) · currentPrices[pair]
6. For each pair, initialize self.encAggregate[pair] to the encryption of 0.
7. Clear self.aggregate

Figure 9: Initialize a round.

**Public Call: ORDER-BOOK.FinalizeCollectPhase**

1. Assert (currentBlock - self.currentRoundStart) ≥ LENCOLLECTPHASE
2. Set self.currentPhase ← reveal

Figure 10: Triggering the end of the collect phase.
Public Call: ORDER-BOOK.RevealTradeValues

**Input:** aggregate: Map(Pair, Amount), proofs: Map(Pair, DecProof)

// To get aggregate and proofs the sender calls DecryptionOracle.Decrypt on self.encAggregate.

1. **Assert** self.currentPhase = reveal
2. For each pair in self.encAggregate:
   - Set π = proofs[pair]
   - Set amount = aggregate[pair]
   - Set encAmount = self.encAggregate[pair]
   - **Assert** DecryptionOracle.Verify(self.encParams, encAmount, amount; π)
     // Assert that aggregate[pair] is the decryption of self.encAggregate[pair].
3. For each element (pair, amount) in self.aggregate:
   Send amount of tokens pair from to SWAP-ENGINE
4. Set self.aggregate = aggregate
5. Set self.currentPhase = trade
6. Call SWAP-ENGINE.Start(aggregate, self.maxBuyPrices)

Figure 11: Reveal the encrypted aggregated trade values.

Public Call: ORDER-BOOK.FinalizeTradePhase

**Input:** sold: Map(Pair, Amount), bought: Map(Pair, Amount)

1. **Assert** caller = SWAP-ENGINE.
2. For each element (pair, amount) in self.aggregate:
   - Set tradedFraction = sold[pair] / self.aggregate[pair]
   - Set price = sold[pair] / bought[pair], (unless tradedFraction = 0, in which case set price = 0)
   - **Initialize** event: TradeEvent
     - event.pair = pair
     - event.tradedFraction = tradedFraction
     - event.round = self.currentRound
     - event.price = price
   - self.AddEvent(event)
3. Call ORDER-BOOK.InitializeRound(self.currentRound + 1)

Figure 12: ORDER-BOOK receives the trading results from SWAP-ENGINE and the round is finalized.
### 6.2.6 Basic Token Note Management

Below we provide implementation of the two basic functionalities of the shielded pool: `ORDER-BOOK.DepositTokens` and `ORDER-BOOK.WithdrawTokens`. The former allows to deposit a note in the shielded pool by sending public coins to the contract. The latter allows to spend a note: withdraw part of its funds to a public account and keep the rest but in a new, separate note.

**Public Call: ORDER-BOOK.DepositTokens**

**Input:** amount: Amount, tokenId: Token, noteHash: Scalar, proof: ZKProof

1. **Assert** the caller has sent value of tokens tokenId along the transaction\(^*\)
2. **Assert** 0 ≤ amount ≤ MAXSUPPLY
3. Set \( x_{\text{DepositTokens}} = (\text{amount}, \text{tokenId}, \text{noteHash}) \)
4. **Assert** ZKP.V(\( R_{\text{DepositTokens}}, x_{\text{DepositTokens}}, \text{proof} \))
5. `self.tokenBag.addLeaf(noteHash)`

\(^*\)In a real implementation this should be done using ERC20 allowance.

![Figure 13: Deposit tokens to create a note.](image)

**Public Call: ORDER-BOOK.WithdrawTokens**

**Input:** valueOut: Amount, tokenId: Token, root: Scalar, proof: ZKProof, tokenNullifier: Scalar, newNoteHash: Scalar

1. **Assert** `self.tokenBag.isHistoricalRoot(root)`
2. **Assert** `self.tokenNullifierSet.nullify(tokenNullifier)`
3. Set \( x_{\text{WithdrawTokens}} = (\text{root}, \text{valueOut}, \text{tokenId}, \text{newNoteHash}, \text{tokenNullifier}) \)
4. **Assert** ZKP.V(\( R_{\text{WithdrawTokens}}, x_{\text{WithdrawTokens}}, \text{proof} \))
5. `self.tokenBag.addLeaf(newNoteHash)`
6. Send `valueOut` tokens `tokenId` to the caller

![Figure 14: Withdraw tokens from a note.](image)

### 6.2.7 Advanced Token Note Management

Apart from the basic functionality of the shielded pool, it is also convenient to give the user more flexibility to manage the notes with additional calls. These are:

- **ORDER-BOOK.MergeNotes** – takes hashes of two notes with the same token, spends them, and creates a new note with value being the sum of values of the spent notes,
- **ORDER-BOOK.SplitNote** – the opposite of `ORDER-BOOK.MergeNotes`, takes one note and splits it into two.
The implementations of the above two methods is straightforward when already given ORDER-BOOK\texttt{DepositTokens} and ORDER-BOOK\texttt{WithdrawTokens}, hence we omit the details.

6.3 User Actions

While Subsection 6.2 provides the description of the ORDER-BOOK contract calls, it might not be immediately clear how does the interaction with the ORDER-BOOK should look like from the user perspective. The tricky part is mostly about keeping track of the various secrets that are necessary to claim the tokens.

6.3.1 Storage

The user’s storage consists of two collections of notes: token notes localNoteSet and order notes localOrderSet. It is necessary to keep track of these notes to be able to claim the underlying funds. In a practical implementation the wallet could keep these locally or in a cloud, or perhaps generate the relevant secrets pseudorandomly from a secret seed, so that they can be recovered without the need to keep lots of data.

1. localNoteSet : Set\texttt{Note} – a set of notes that the user owns, i.e. their hashes are held in ORDER-BOOK.tokenBag

2. localOrderSet : Map\texttt{OrderId, OrderNote} – a mapping of order ids into order notes.
### 6.3.2 Interactions

**User Action:** CreateNote

<table>
<thead>
<tr>
<th>Input:</th>
<th>tokenId : Token, amount : Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Sample elements tokenTrapdoor, tokenNullifier ← Scalar.</td>
</tr>
<tr>
<td></td>
<td>2. Initialize note : Note</td>
</tr>
<tr>
<td></td>
<td>• note.tokenId = tokenId,</td>
</tr>
<tr>
<td></td>
<td>• note.amount = amount,</td>
</tr>
<tr>
<td></td>
<td>• note.tokenTrapdoor = tokenTrapdoor,</td>
</tr>
<tr>
<td></td>
<td>• note.tokenNullifier = tokenNullifier.</td>
</tr>
<tr>
<td></td>
<td>3. Set noteHash = Hash(note)</td>
</tr>
<tr>
<td></td>
<td>4. Set w = (note)</td>
</tr>
<tr>
<td></td>
<td>5. Set x = (noteHash, tokenId, amount)</td>
</tr>
<tr>
<td></td>
<td>6. Set proof ← ZKP.P(RDepositTokens, x, w)</td>
</tr>
<tr>
<td></td>
<td>7. Sign transaction tx = ORDER-BOOK.DebitTokens(noteHash, tokenId, amount, proof), which has attached amount of token tokenId.</td>
</tr>
<tr>
<td></td>
<td>8. Send tx and wait until it has been processed on chain.</td>
</tr>
<tr>
<td></td>
<td>9. Assert tx executed without errors on chain.</td>
</tr>
<tr>
<td></td>
<td>10. Add the note to the local storage: self.localNoteSet.Add(note)</td>
</tr>
</tbody>
</table>

Figure 15: Create a new note.
User Action: **CreateOrder**

**Input:** pair: Pair, amount: Amount, maxPrice: Price

1. Retrieve note from the localNoteSet such that note.amount = amount and note.pair = pair.
   
   // If there is no such note the user is supposed to use ORDER-BOOK.MergeNotes
   // and ORDER-BOOK.SplitNote to create one.

2. Set scaledAmount = \[\text{amount}/N\]

3. Sample orderId \(\leftarrow \$\) Scalar

4. Sample orderTrapdoor, orderNullifier \(\leftarrow \$\) Scalar

5. Set encPKey = ORDER-BOOK.encPKey

6. Compute the encryption of amount as

\[
\text{encAmount} = \text{DecryptionOracle.Encrypt}(\text{encPKey}, \text{scaledAmount}; r)
\]

with \(r \leftarrow \$\) Scalar

7. Initialize orderNote: OrderNote

   - orderNote.orderId = orderId
   - orderNote.pair = pair
   - orderNote.scaledAmount = scaledAmount
   - orderNote.orderTrapdoor = orderTrapdoor
   - orderNote.orderNullifier = orderNullifier

8. Set orderHash = \text{Hash}(\text{orderNote})

9. Set noteHash = \text{Hash}(\text{note})

10. Set \((\text{root, merkleProof}) = \text{ORDER-BOOK.tokenBag.generateProof}(\text{noteHash})\)

11. Set \(x = (\text{root, noteHash, orderHash, encAmount})\).

12. Set \(w = (\text{note, orderNote, path, r}).\)

13. Set proof = ZKP.P.\(\text{R}_{\text{NewOrder}}, x, w)\)

14. Sign transaction

\[
\text{tx} = \text{ORDER-BOOK.NewOrder}(\text{orderId, pair, maxPrice, root, noteHash, orderHash, encAmount, proof, tokenNullifier})
\]

15. Send tx and wait until it has been processed on chain.

16. Add the order to the local storage: \texttt{self.localOrderSet[orderId] = orderNote}

---

**Figure 16:** Create a new order.
<table>
<thead>
<tr>
<th>User Action: <strong>CancelOrder</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> orderId : OrderId</td>
</tr>
<tr>
<td>1. Retrieve orderNote = localOrderSet[orderId] from the local storage</td>
</tr>
<tr>
<td>2. Set orderHash = Hash(orderNote)</td>
</tr>
<tr>
<td>3. (root, merkleProof) = ORDER-BOOK.orderBag.generateProof(orderHash)</td>
</tr>
<tr>
<td>4. Set x = (root, orderId)</td>
</tr>
<tr>
<td>5. Set w = (orderNotePath, orderNote)</td>
</tr>
<tr>
<td>6. Set proof = ZKP.P(R\textsubscript{CancelOrder}, x, w)</td>
</tr>
<tr>
<td>7. Sign transaction tx = ORDER-BOOK.\texttt{CancelOrder}(orderId, root, proof)</td>
</tr>
<tr>
<td>8. Send tx and wait until it has been processed on chain.</td>
</tr>
</tbody>
</table>

Figure 17: Cancel an order.
User Action: ClaimCancelled

Input: orderId : OrderId,

1. Retrieve orderNote = localOrderSet[orderId] from the local storage.
2. Set order = ORDER-BOOK.ordersSet[orderId]
3. Assert order.isCancelled = True, order.lastBatch = None, and order.fillRatio < 1.
4. Set cancelOrderNullifier = Hash(orderNote.orderNullifier, cancel)
5. Set orderAmount = orderNote.scaledAmount · N
   // Conversion from ScaledAmount to Amount
6. Set valueClaimed = orderAmount · (1 − fillRatio)
   // Cf. Section 2.3 for an explanation of how this expression is computed under the hood
7. Sample elements tokenTrapdoor, tokenNullifier ← Scalar.
8. Initialize note : Note
   • note.tokenId = pair.from,
   • note.amount = valueClaimed,
   • note.tokenTrapdoor = tokenTrapdoor,
   • note.tokenNullifier = tokenNullifier.
9. Set noteHash ← Hash(note)
10. Set orderHash = Hash(orderNote)
11. (rootOrderBag, orderNotePath) = ORDER-BOOK.orderBag.generateProof(orderHash)
12. Set x = (cancelOrderNullifier, rootOrderBag, orderId, noteHash, fillRatio)
13. Set w = (orderNote, note, orderNotePath)
14. proof ← ZKP.P(RClaimCancelled, x, w)
15. Sign transaction

\[
\text{tx} = \text{ORDER-BOOK.ClaimCancelled}(\text{cancelOrderNullifier, rootOrderBag, }
\text{orderId, noteHash, proof})
\]

16. Send tx and wait until it has been processed on chain.
17. Assert tx executed without errors on chain.
18. Add the note to the local storage: self.localNoteSet.Add(note)

Figure 18: Claim a cancelled order.
User Action: ClaimSwapped

Input: orderId : OrderId, orderInBatch : OrderInBatchEvent, trade : TradeEvent

// The events are public on chain, so the user can fetch them if needed.

1. Set orderNote = localOrdersSet[orderId]
2. Set round = orderInBatch.round
3. Check that the events orderInBatch and trade express the fact that order orderId was included and traded in round round.
4. Set \( \text{rootOrderBag, orderNotePath} \) = \( \text{ORDER-BOOK.orderBag.generateProof(Hash(orderNote))} \)
5. Set \( \text{rootEventLog, merklePathBatch} \) = \( \text{ORDER-BOOK.eventLog.generateProof(Hash(orderInBatch))} \)
6. Set \( \text{rootEventLog, merklePathTrade} \) = \( \text{ORDER-BOOK.eventLog.generateProof(Hash(trade))} \)
7. Set swapOrderNullifier = Hash(orderNote.orderNullifier, round)
8. Set amountTraded = \( (\text{orderNote.scaledAmount} \cdot \text{orderInBatch.scaledOrderFraction}) \cdot \text{trade.fractionTraded} \)
9. Set valueClaimed = amountTraded \cdot \text{trade.price}
10. Initialize note : Note:
    • note.tokenId = orderNote.pair.to
    • note.amount = valueClaimed
    • note.tokenTrapdoor = tokenTrapdoor, randomly sampled from Scalar
    • note.tokenNullifier = tokenNullifier, randomly sampled from Scalar
11. Set noteHash = Hash(note)
12. Set \( x = (\text{swapOrderNullifier, rootOrderBag, rootEventLog, noteHash}) \)
13. Set \( w = \left( \begin{array}{c}
          \text{orderInBatch, trade,} \\
          \text{orderNotePath, merklePathBatch, merklePathTrade}
        \end{array} \right) \)
14. Set \( \text{proof} = \text{ZKP.P}(R_{\text{ClaimSwapped}}, x, w) \)
15. Sign transaction

\[ \text{tx} = \text{ORDER-BOOK.ClaimSwapped}(\text{swapOrderNullifier, rootOrderBag, rootEventLog, noteHash, proof}) \]

16. Send \( \text{tx} \) and wait until it has been processed on chain.
17. Assert \( \text{tx} \) executed without errors on chain.
18. Add the note to the local storage: self.localNoteSet.Add(note)

Figure 19: Claim a swapped order.
6.4 Updaters

In order for the system to work correctly we need one or more parties to function as *Updaters*. This is a role we distinguish in the system, even though there are no permissions necessary to act in this role – in fact any user of the blockchain can be an updater. The role of an updater is quite pragmatic: its goal is to make the contract progress with rounds in a timely fashion – thus, most of all, trigger the start of new phases and rounds. The reason we need updaters at all is because of the execution model of smart contracts – a contract can change its state only when triggered by a transaction. On the other hand, a contract cannot schedule a state change based on some conditions being met in the future (like a particular block number or so). Updaters are parties who are responsible for triggering the actions.

Before we list the particular responsibilities of updaters, let us briefly discuss the incentives behind such an activity. Note that updaters are expected to send some transactions, and thus they bear the cost of transactions fees. An important question to ask is then, why would they do that, if there is a clear cost, but no benefits? The answer is twofold:

- Regular, frequent users (perhaps market makers) might be interested in acting as updaters if the profit they make on trading on Common justifies the costs of running an updater. They simply have an incentive to keep Common running because they directly benefit from this fact.
- One could add an incentive mechanism to Common where updater actions would yield monetary rewards outweighing the fee costs. One possible source of the rewards for updaters could be trading fees that Common could collect (which it doesn’t in the current version).

6.4.1 Updater Responsibilities

- **Finalize phases.** While the rounds and phases are completely determined by block height, the updates must still trigger some such events using transactions. Here, the updater is expected to call `ORDER-BOOK.FinalizeCollectPhase` when the phase is over (according to block height).

- **Reveal batch.** Once in the *reveal* phase, the updater is expected to reach out to the Decryption Oracle and fetch the plain text values of traded values, and then call `ORDER-BOOK.RevealTradeValues`.

- **Include orders in a batch.** In each round there are deterministic requirements on which orders should be included. The responsibility of the updater is to find all orders that satisfy these requirements and call `ORDER-BOOK.PlaceOrderInBatch` on each.

- **Update orders.** Whenever an order has been traded in a batch it enters a state in which it cannot be added to a batch again (because data in `ORDER-BOOK.ordersSet` is not up-to-date) and the updaters are expected to call `ORDER-BOOK.UpdateOrder` on each such order.
7 Swap Engine

This section is devoted to a formal description of the SWAP-ENGINE contract. For a high level description we refer to Section 4.4.

7.1 Storage

We list all the storage items of ORDER-BOOK along with their types. In the pseudocode a given storage item, like sold, is referred to as self.sold because self is the SWAP-ENGINE itself.

1. initialFrom : Map (Pair, Amount) – a mapping specifying for each trading pair \((A, B)\) how many \(A\) tokens should be sold for \(B\) tokens.
2. sold : Map (Pair, Amount) – a mapping associating a pair \((A, B)\) to the amount of token \(A\) that the SWAP-ENGINE has already sold (this includes the amount of token sold during the internal matching).
3. bought : Map (Pair, Amount) – a mapping associating a pair \((A, B)\) to the amount of token \(B\) that the SWAP-ENGINE has received in return for sold[\((A, B)\)] token \(A\) (includes the amount of token received during the internal matching).
4. startAuctionPrices : Map (Pair, Price) – a mapping associating a pair to the price that is used when the Dutch auction is initiated. If a token pair \((A, B)\) is associated with a price \(p\) then it means that \(p \cdot x\) token \(A\) is worth \(x\) token \(B\).
5. maxBuyPrices : Map (Pair, Price) – a mapping associating a pair \((A, B)\) to the maximum price for which the SWAP-ENGINE can buy token \(B\) in return for token \(A\). The Dutch auction will begin with price of buying token \(B\) (in return for \(A\)) from the side of SWAP-ENGINE equal to the minimum of the oracle price and maxBuyPrices. Note that maxBuyPrices is determined based on the prices from the query of the ORDER-BOOK to the price oracle at the beginning of the round. At the moment when SWAP-ENGINE queries again the price oracle, these prices may be already higher than the maxBuyPrices. The price of selling token \(B\) in return for \(A\) from the side of the Market Makers (and buying token \(B\) from the side of the \(A\)) will increase until all the available amount has been sold or the length of the auction reaches AUCTIONLENGTH.
6. timeStart : Int – stores the block number when the current auction has started.
7.2 Calls

Public Call: SWAP-ENGINE.Start

<table>
<thead>
<tr>
<th>Input:</th>
<th>initialFrom : Map (Pair, Amount), maxBuyPrices : Map (Pair, Price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Assert caller = ORDER-BOOK.</td>
<td></td>
</tr>
<tr>
<td>2. Initialize for every pair:</td>
<td></td>
</tr>
<tr>
<td>• self.sold[pair] = 0</td>
<td></td>
</tr>
<tr>
<td>• self.bought[pair] = 0</td>
<td></td>
</tr>
<tr>
<td>• self.initialFrom[pair] = 0</td>
<td></td>
</tr>
<tr>
<td>• self.startAuctionPrices[pair] = 0</td>
<td></td>
</tr>
<tr>
<td>• self.maxBuyPrices[pair] = 0</td>
<td></td>
</tr>
<tr>
<td>3. Set self.initialFrom = initialFrom</td>
<td></td>
</tr>
<tr>
<td>4. Set self.maxBuyPrices = maxBuyPrices</td>
<td></td>
</tr>
<tr>
<td>5. For each unordered pair of tokens ( {A, B} ):</td>
<td></td>
</tr>
<tr>
<td>SWAP-ENGINE.InternalMatching((A, B))</td>
<td></td>
</tr>
<tr>
<td>6. For each pair do:</td>
<td></td>
</tr>
<tr>
<td>• oraclePrice = PRICE-ORACLE.QueryPrices(pair)</td>
<td></td>
</tr>
<tr>
<td>• self.startAuctionPrices[pair] = min(oraclePrice, self.maxBuyPrices[pair])</td>
<td></td>
</tr>
<tr>
<td>7. Initialize self.time_start = currentBlock</td>
<td></td>
</tr>
</tbody>
</table>

Figure 20: Receives tokens from the ORDER-BOOK, performs maximum possible internal matching, and initiates Dutch Auction for selling the remaining amount. Called by ORDER-BOOK.
### Public Call: SWAP-ENGINE.ParticipateAuction

**Input:** $(A, B): \text{Pair, buyAmountA: Amount, price: Price}$

// A Market Maker makes this call to buy buyAmountA tokens A at price at most \( \frac{1}{\text{price}} \).
// The price parameter is useful to make sure the transaction executes only
// if the price at the current block is compatible with the sender’s request.

1. Assert \( (\text{currentBlock} - \text{self.timeStart}) \leq \text{AUCTIONLENGTH} \)
2. Set \( p = \text{SWAP-ENGINE.DutchAuctionPrice(\text{currentBlock}, (A, B))} \)
3. Assert \( p \geq \text{price} \)
4. Assert that caller has sent to SWAP-ENGINE \( \frac{\text{buyAmountA}}{\text{price}} \) tokens B.
5. Set \( \text{tradedAmountA} = \min(\text{self.initialFrom}[(A, B)] - \text{self.sold}[(A, B)], \text{buyAmountA}) \)
   // The buyAmountA specifies the maximum amount that the caller wants to buy
   // If there is not enough available, all will be sold.
6. Set \( \text{tradedAmountB} = \frac{\text{tradedAmountA}}{p} \)
7. Send to the caller \( \text{tradedAmountA} \) tokens A and refund the B tokens that were not traded, i.e., an amount of \( \frac{\text{buyAmountA}}{\text{price}} - \text{tradedAmountB} \)
8. \( \text{self.sold}[(A, B)] = \text{self.sold}[(A, B)] + \text{tradedAmountA} \)
9. \( \text{self.bought}[(A, B)] = \text{self.bought}[(A, B)] + \text{tradedAmountB} \)

Figure 21: Participate in the Dutch Auction for specific pair and amount to buy.

### Internal Call: SWAP-ENGINE.DutchAuctionPrice

**Input:** \( \text{blockNum, (A, B): Pair} \)

1. Set \( \alpha = \frac{\text{blockNum} - \text{self.timeStart}}{\text{AUCTIONLENGTH}} \)
2. Return \( (1 - \alpha) \cdot \text{startAuctionPrices}[(A, B)] + \alpha \cdot \text{maxBuyPrices}[(A, B)] \)

Figure 22: Computes the price of the Dutch auction at a particular block height.
Internal Call: SWAP-ENGINE.InternalMatching

Input: \((A, B): \text{Pair}\)

1. \textbf{Assert } \(\text{self.maxBuyPrices}[(B, A)]^{-1} \leq \text{self.maxBuyPrices}[(A, B)]\)
   
   // The above condition guarantees existence of a common matching price.

2. \textbf{Set } \(p = \sqrt{\frac{\text{self.maxBuyPrices}[(A, B)]}{\text{self.maxBuyPrices}[(B, A)]}}\)

3. \textbf{self.sold}[(B, A)] = \min\left(\frac{\text{self.initialFrom}[(A, B)]}{p}, \text{self.initialFrom}[(B, A)]\right)

4. \textbf{self.bought}[(A, B)] = \text{self.sold}[(B, A)]

5. \textbf{self.sold}[(A, B)] = p \cdot \text{self.sold}[(B, A)]

6. \textbf{self.bought}[(B, A)] = \text{self.sold}[(A, B)]

Figure 23: Computes a common price and performs internal matching if possible.

Public Call: SWAP-ENGINE.FinalizeAuction

Input:

1. \textbf{Assert } (\text{currentBlock} - \text{self.time.start}) > \text{AUCTIONLENGTH}\)

2. Transfer all the traded and untraded tokens back to ORDER-BOOK

3. Call ORDER-BOOK.FinalizeTradePhase(\text{self.sold}, \text{self.bought})

4. \textbf{clear } \text{self.initialFrom}, \text{self.startAuctionPrices}, \text{self.maxBuyPrices}

5. \textbf{clear } \text{self.sold}, \text{self.bought}, \text{self.time.start}.
   
   // It is important that the maps include zeros after that step in order to
   // prevent Market makers whose transaction was included in a later round
   // to trade in phases different from trade. (step 5 in SWAP-ENGINE.ParticipateAuction).

*Alternatively one can allow finishing the auction in case everything is sold.

Figure 24: Finalize the auction and pass the results to ORDER-BOOK.

7.3 User Actions

Market makers will monitor the Dutch auction with the purpose to exchange tokens in a
price they find attractive. When they agree with the price, then they send a transaction
that triggers SWAP-ENGINE.ParticipateAuction. Note that as the Dutch auction pro-
gresses, the price becomes more favourable to the market makers. However, if a market
maker waits too long, they take the risk of other market makers buying all the avail-
able amount. Moreover, when the Market makers send their transaction, they include
as input the minimum price at which they want to sell the specified token. This ensures
that even if their transaction is included in the \textit{trade} phase of a later round, where the
prices for the pairs may differ, they will not trade at a worse price than what they have
specified; in that case their transaction will not be executed if the current price does not
satisfy their limit. Note that if their transaction is included in the collect, reveal phases of a later round, then their trade will not be executed, because the relevant maps that specify the available amount for trading will contain 0.

7.4 Updaters

Updaters are supposed to call the SWAP-ENGINE.FinalizeAuction in order for the Dutch auction to become finalized and for the ORDER-BOOK.FinalizeTradePhase to get called. Note that when the SWAP-ENGINE.FinalizeAuction is triggered, it checks that the length of the Dutch auction has reached the AUCTIONLENGTH. Thus, we do not need to trust the updater, as they cannot terminate the auction early. Also, even if no updater triggers SWAP-ENGINE.FinalizeAuction right away the Dutch auction will not accept any more requests from the market makers (as SWAP-ENGINE.ParticipateAuction asserts that the length of the auction has not exceeded AUCTIONLENGTH).

8 Relations

In this section we formally describe the relations used in COMMON. Each relation is specified using the following template.

```
<table>
<thead>
<tr>
<th>Relation: R_{example}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: // Instance</td>
</tr>
<tr>
<td>Witness:   // Witness</td>
</tr>
<tr>
<td>Constraints:</td>
</tr>
<tr>
<td>• // Constraints involving the instance and witness</td>
</tr>
</tbody>
</table>
```

Figure 25: Template for specifying constraint bundles

When it comes to specifying the list of witnesses in our relations, we list just the essential ones, and, for ease of presentation, we don’t include temporary variables and “hint” variables in case they are computable from other witnesses.

Some of these relations share some subset of constraints. Because of this, and to lighten the presentation, we informally introduce the concept of “constraint bundle”. These are simply collections of constraints that can be included inside the constraint section of relations. A constraint bundle places no assumptions on what parts of the elements involved are public or private: in practice, this is taken care by the relation where the constraint bundle is included. Constraint bundles are specified with the following template.
8.1 Constraint bundle: Merkle tree membership

With the goal of fixing notation, in this subsection we define a constraint enforcing Merkle tree membership. However, since this is a standard concept, we omit most of the details.

In short, the constraint enforces that, for public inputs a Merkle tree root \( \text{root} \) and a purported leaf \( \text{leaf} \) in the tree, there exist a path \( \text{path} \) from \( \text{leaf} \) to \( \text{root} \) in the tree. The witness for \( (\text{root}, \text{leaf}) \) is such path (or, more precisely, the hashes in the nodes of the path), together with the childs of each node in the path. Abusing the terminology, we denote such witness by \( \text{path} \).

8.2 Constraint bundle: correct link between order and note

These constraints enforce that the contents of a note \( \text{note} \) and an order note \( \text{orderNote} \) are “consistent”. Precisely, it enforces that the token in \( \text{note} \) coincides with the token being sold in \( \text{orderNote} \), and the amount \( \text{scaledAmount} \) of tokens in \( \text{orderNote} \) is precisely \( \lfloor \text{note.amount}/M \rfloor \).
8.3 Constraint bundle: correct token nullifier

The following constraint bundle enforces that the nullifier of a note coincides with a specific nullifier.

<table>
<thead>
<tr>
<th>Constraint: CONSTR\textsubscript{tokenNullifier}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: note, tokenNullifier</td>
</tr>
<tr>
<td><strong>Constraints</strong>:</td>
</tr>
<tr>
<td>• note.tokenNullifier = tokenId</td>
</tr>
</tbody>
</table>

8.4 New order

This relation enforces that an order \texttt{orderNote} created while executing the command \texttt{NewOrder} is created correctly. More precisely, it checks that:

- Both \texttt{note} and \texttt{orderNote} (which are secret) hash into publicly known values.
- The user has a note \texttt{note} in \texttt{tokenBag}.
- The contents of \texttt{note} and \texttt{orderNote} are consistent, as enforced by the constraint bundle \texttt{CONSTR\textsubscript{link}}.
- The token note \texttt{note} contains a publicly specified nullifier \texttt{tokenNullifier}.
- The token amount \texttt{scaledAmount} in \texttt{orderNote} is the plaintext corresponding to a public ciphertext \texttt{encAmount}. 
Relation: $R_{\text{newOrder}}$

Instance: root, noteHash, tokenNullifier, orderHash, encAmount, encPKey

Witness: note, orderNote, path, $r$

// root and path are supposed to be, respectively, a root of tokenBag,
// and the path in tokenBag from noteHash to root.

Constraints:

- orderHash = Hash(orderNote)
- noteHash = Hash(note)
- $(\text{root}, \text{noteHash}, \text{path}) \in \text{CONSTR}_{\text{Merkle-tree}}$
  // Ensures that noteHash is a leaf of the tokenBag Merkle tree
- $(\text{note}, \text{orderNote}) \in \text{CONSTR}_{\text{link}}$
  // Ensures that tokenId in note and the token being sold in orderNote are the same
  // It also enforces that the respective token amounts are “consistent”
- $(\text{note}, \text{tokenNullifier}) \in \text{CONSTR}_{\text{tokenNullifier}}$
  // Verifies that note contains the public nullifier tokenNullifier
- encAmount = DecryptionOracle.Encrypt(encPKey, orderNote.scaledAmount, $r$)
  // In Section 9 we discuss how to write the above as a circuit.

Figure 29: Relation enforcing the correct creation of an order.

8.5 Constraint bundle: Order ownership

Intuitively, these constraints will be used show that the prover is the “owner” of a secret order note orderNote whose order Id is public. This is attained by enforcing that the hash of orderNote belongs to orderBag; and that the field orderId in orderNote matches the public order Id.

Constraint: $\text{CONSTR}_{\text{NoteOrderOwnership}}$

Input: rootOrderBag, orderId, orderNotePath, orderNote

Constraints:

- Set orderHash = Hash(orderNote)
- $(\text{rootOrderBag}, \text{orderHash}, \text{orderNotePath}) \in \text{CONSTR}_{\text{Merkle-tree}}$
- orderNote.orderId = orderId

Figure 30: Constraints enforcing that the prover is the “owner” of an order note.

8.6 Cancel order

To cancel an order, it is enough to show ownership of the order. Hence we let $R_{\text{CancelOrder}}$ be $\text{CONSTR}_{\text{NoteOrderOwnership}}$ with appropriately specified public inputs and witnesses. Precisely:
Figure 31: Relation enforcing all necessary checks to cancel an order.

8.7 Claim cancelled order

To claim a cancelled order for a certain value \texttt{valueClaimed}, the user:

- Shows it is the owner of the order note, as enforced by the constraints given in \texttt{CONSTRNoteOrderOwnership}.
- Shows that \texttt{note} hashes into a publicly known value.
- Shows that the contents of \texttt{note} and \texttt{orderNote} are “consistent”, i.e. that

\[
\texttt{note}.\texttt{tokenId} = \texttt{orderNote}.\texttt{pair}.\texttt{from}.
\]

- Shows that the token amount in \texttt{note} is the result of multiplying the initial order’s amount by the value \(N \cdot (1 - \texttt{orderNote}.\texttt{fillRatio})\). Here we use the multiplication definition from Section 2.2, which is enforced using the constraint bundle \texttt{CONSTRFixedPointAmountMul} from Section 8.11. Here we multiply by \(N\) here since \texttt{orderNote} holds token amounts \texttt{scaledAmount}, which represent the floor division of an actual token amount by the scaling factor \(N\). Intuitively, multiplying by a value \texttt{scaledAmount} by \(N\) “undoes the scaling”.
- Shows that the public order nullifier is correctly computed.

Figure 32: Relation enforcing all necessary checks allowing a user to claim the tokens left in a cancelled order.
8.8 Claim swap

To claim a partially swapped order, the user:

- Shows it knows the contents of an order note `orderNote` belonging to the order bag.
- Shows it knows two events, `orderInBatch` and `trade` in the event log.
- Shows that all of `orderNote` and `orderInBatch` refer to the same order Id. Similarly, the user shows that `orderInBatch` and `trade` refer to the same round number and to the same pair of tokens.
- Shows the public order nullifier is correctly computed.
- Shows that it knows the contents of a note `note` whose hash is the public value `noteHash`.
- Shows that the token Id in `note` is the token being bought in `orderNote` (i.e. `note.tokenId = orderNote.pair.to`)
- Shows that the value `amount` in `note` is consistent with the amount of tokens traded in `trade`. Precisely,

\[
\text{note.amount = } \\
((\text{orderNote.scaledAmount} \cdot \text{orderInBatch.scaledOrderFraction}) \\
\cdot \text{trade.fractionTraded}) \cdot \text{trade.price}.
\]
Relation: \( R_{\text{ClaimSwapped}} \)

Instance: \((\text{swapOrderNullifier}, \text{rootOrderBag}, \text{rootEventLog}, \text{noteHash})\)
Witness: \((\text{orderNote}, \text{note}, \text{orderInBatch}, \text{trade}, \text{orderNotePath}, \text{merklePathBatch}, \text{merklePathTrade})\)

Constraints:
- \((\text{rootOrderBag}, \text{Hash}(\text{orderNote}), \text{orderNotePath}) \in \text{CONSTR}_{\text{Merkle-tree}}\)
- \((\text{rootEventLog}, \text{Hash}(\text{orderBag}), \text{merklePathBatch}) \in \text{CONSTR}_{\text{Merkle-tree}}\)
- \((\text{rootEventLog}, \text{Hash}(\text{trade}), \text{merklePathTrade}) \in \text{CONSTR}_{\text{Merkle-tree}}\)
- \(\text{orderNote}.\text{orderId} = \text{orderInBatch}.\text{orderId}\)
- \(\text{orderInBatch}.\text{round} = \text{trade}.\text{round}\)
- \(\text{orderInBatch}.\text{pair} = \text{trade}.\text{pair}\)
- \(\text{swapOrderNullifier} = \text{Hash}(\text{orderNote}.\text{orderNullifier}, \text{event}.\text{round})\)
- \(\text{noteHash} = \text{Hash}(\text{note})\)
- \(\text{note}.\text{tokenId} = \text{order}.\text{pair}.\text{to}\)

The next three constraints enforce that \(\text{note}.\text{amount} = ((\text{orderNote}.\text{scaledAmount} \cdot \text{orderInBatch}.\text{scaledOrderFraction}) \cdot \text{trade}.\text{fractionTraded}) \cdot \text{trade}.\text{price}\) (cf. Section 2.2). The values \(\text{aux}_1, \text{aux}_2\) are auxiliary witness entries which we omit in the witness declaration above.

- \((\text{orderNote}.\text{scaledAmount}, \text{orderInBatch}.\text{scaledOrderFraction}, \text{aux}_1) \in \text{CONSTR}_{\text{FixedPointAmountMul}}\)
- \((\text{aux}_1, \text{trade}.\text{fractionTraded}, \text{aux}_2) \in \text{CONSTR}_{\text{FixedPointAmountMul}}\)
- \((\text{aux}_2, \text{trade}.\text{price}, \text{note}.\text{amount}) \in \text{CONSTR}_{\text{FixedPointAmountMul}}\)

Figure 33: Relation enforcing all necessary checks allowing a user to claim an amount of traded tokens.

8.9 Deposit tokens

This relation is used when a user deposits a tokens (in the form of a note) in the order book. The relation enforces that a private note \text{note} (with publicly specified hash) refers to a public token Id \text{tokenId} and contains a publicly specified amount of tokens \text{amount}.
8.10 Withdraw tokens

This relation is used when a user attempts to withdraw tokens from the order book. The user must prove that:

- It knows the contents of a note `note` belonging to the token bag.
- This note contains a publicly specified nullifier.
- It knows the contents of a “new” note `newNote` whose hash is publicly known. This new note will hold the tokens from `note` that are not withdrawn.
- Both `note` and `newNote` contain the same type of tokens, i.e.

  \[ \text{note}.\text{tokenId} = \text{newNote}.\text{tokenId} \]

- The amount of tokens in `note` is equal to the amount of tokens in `newNote` plus the amount of tokens `valueOut` being withdrawn. Moreover, the latter is not larger than `MAXSUPPLY`.

Relation: \(R_{\text{WithdrawTokens}}\)

| Instance: root, valueOut, tokenId, newNoteHash, tokenNullifier |
| Witness: noteHash, path, note, newNote |
| Constraints: |
  - (root, Hash(note), path) \(\in\) \(\text{CONSTR}_{\text{Merke-tree}}\)
  - (note, tokenNullifier) \(\in\) \(\text{CONSTR}_{\text{tokenNullifier}}\)
  - newNoteHash = Hash(newNote)
  - note.tokenId = tokenId and newNote.tokenId = tokenId
  - 0 \(\leq\) newNote.value \(\leq\) MAXSUPPLY
  - newNote.value + valueOut = note.value

Figure 35: Relation used when withdrawing tokens in the order book.
8.11 Constraint bundles: non-standard arithmetics

8.11.1 Fixed point arithmetic

In this section we use Lemma A.1 from Appendix A to define constraints that enforce correct computations between values of type `FixedPoint`. The main technical result we need is the following:

**Lemma 8.1.** Let \( a, b, c \in \mathbb{N} \cup \{0\} \) be three natural numbers (possibly zero), with \( b \neq 0 \). The following holds

\[
\left\lfloor \frac{a}{b} \right\rfloor = c \text{ if and only if } 0 \leq a - bc < b.
\]

**Proof.** Suppose \( \left\lfloor \frac{a}{b} \right\rfloor = c \) and write \( \frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor + \varepsilon \) for some \( n \in \mathbb{N} \cup \{0\} \) and \( 0 \leq \varepsilon < 1 \). Then \( a - bc = b\left\lfloor \frac{a}{b} \right\rfloor + b\varepsilon - bc = b\varepsilon \), and clearly \( 0 \leq b\varepsilon < b \). Conversely, suppose \( 0 \leq a - bc < b \). Then \( 0 \leq a/b - c = n + \varepsilon - c < 1 \), where \( n = \left\lfloor a/b \right\rfloor \) and \( 0 \leq \varepsilon < 1 \). Now, being an integer, \( n - c \) must be 0, since otherwise \((n - c) + \varepsilon\) would be outside the range \([0, 1)\). Hence \( n = \left\lfloor a/b \right\rfloor = c \). \( \square \)

For convenience, we recall here the definitions of addition and multiplication between values of type `FixedPoint` (see Section 2.2.1):

\[
y_1 + y_2 = \begin{cases} 
y_1 + y_2 & \text{if } y_1 + y_2 < B, \\
\text{Err} & \text{otherwise}
\end{cases}
\]

\[
y_1 \cdot y_2 = \begin{cases} 
\left\lfloor \frac{y_1 y_2}{M} \right\rfloor & \text{if } y_1 y_2 < B, \\
\text{Err} & \text{otherwise}
\end{cases}
\]

In views of this definition and of Lemma 8.1 we can define constraints for the above operations as

<table>
<thead>
<tr>
<th>Constraint: CONSTR\textsubscript{FixedPtAdd}</th>
<th>Constraint: CONSTR\textsubscript{FixedPtMul}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ((y_1, y_2, y_3))</td>
<td><strong>Input:</strong> ((y_1, y_2, y_3))</td>
</tr>
<tr>
<td><strong>Constraints:</strong></td>
<td><strong>Constraints:</strong></td>
</tr>
<tr>
<td>• (y_3 = y_1 + y_2),</td>
<td>• (0 \leq y_1 y_2 - y_3 &lt; M)</td>
</tr>
<tr>
<td>• (y_1 + y_2 &lt; B).</td>
<td>• (y_1 \cdot y_2 &lt; B).</td>
</tr>
</tbody>
</table>

Figure 36: Constraint bundles enforcing correctness of the addition and multiplication operations between values of type `FixedPoint` (as defined in Section 2.2).

In Figure 36 the elements \(y_1, y_2, y_3, M, B\) above are understood as elements from \(F\), and all operations are field operations. The inequality \(<\) is understood (abusing the notation) as inequality of natural numbers in the interval \([0, |F| - 1]\).

Note that, due to Lemma 8.1, the following holds for all \(y_1, y_2, y_3 \in \text{FixedPoint}\),

\[
y_1 + y_2 = y_3 \iff \text{CONSTR}\textsubscript{FixedPtAdd}(y_1, y_2, y_3),
\]

\[
y_1 \cdot y_2 = y_3 \iff \text{CONSTR}\textsubscript{FixedPtMul}(y_1, y_2, y_3).
\]
8.11.2 Arithmetic between values of type Amount and FixedPoint

Similarly as in the previous section, we want to write a constraint that enforces correct multiplication of values of type Amount and FixedPoint (see Section 2.2.3). Recall that the given \( a, y \) of type Amount and FixedPoint, we define \( a \cdot y \) as

\[
a \cdot y = \begin{cases} 
\lfloor \frac{a \cdot y}{M} \rfloor & \text{if } a \cdot y \leq B, \\
\text{Err} & \text{otherwise}
\end{cases}
\]

Using a similar rationale as in the previous section, we define

<table>
<thead>
<tr>
<th>Constraint: CONSTR$\text{FixedPointAmountMul}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ((a, y, y'))</td>
</tr>
<tr>
<td><strong>Constraints:</strong></td>
</tr>
<tr>
<td>• (0 \leq a \cdot y - y' &lt; M)</td>
</tr>
<tr>
<td>• (a \cdot y &lt; B)</td>
</tr>
</tbody>
</table>

Figure 37: Constraint bundle enforcing correctness of the multiplication operation between a value of type amount and a value of type FixedPoint (as defined in Section 2.2).

Then, for values \(a, y, y'\) of types Amount, FixedPoint, Amount, respectively, we have

\[
a \cdot y = y' \iff (a, y, y'; \emptyset) \in \text{CONSTRFixedPointAmountMul}.
\]

9 Decryption Oracle

9.1 Functionality

Decryption Oracle is an abstraction layer representing an additively homomorphic encryption scheme along with a party responsible for decrypting ciphertexts when instructed by the on-chain contract ORDER-BOOK. This functionality comprises five subprocedures.

1. **DecryptionOracle.KGen** – takes public parameters \(pp\) and creates encryption, decryption, and verification keys. In practice, Decryption Oracle.KGen is triggered when COMMON is initialized so as to save the suitable public keys in the ORDER-BOOK contract. Parameters \(pp\) determine, e.g., the groups that the underlying encryption scheme uses, the maximal value of the plaintext \(pp\).MAXSUPPLY, the maximal value of plaintext’s chunk \(pp\).MAXENC (which, for the sake of simplicity, is a power of 2), secret key type \(pp\).SecretKey, public key type \(pp\).PublicKey, verification key type VerificationKey and message and ciphertext spaces \(pp\).Message, \(pp\).AHCipherText.

2. **DecryptionOracle.Encrypt** – takes the public key \(pk\) generated in the procedure DecryptionOracle.KGen, a message \(m \in [0, \text{MAXSUPPLY}]\) and possibly randomness \(r\), and encrypts this message into a ciphertext \(c : AHCipherText\).
3. **DecryptionOracle.Add** – adds two ciphertexts to form a new ciphertext. We require additive homomorphism here.

4. **DecryptionOracle.Decrypt** – checks that the aggregated orders in the current round of COMMON are ready to be decrypted, and if this is the case, it decrypts them. Additionally, **DecryptionOracle.Decrypt** outputs a succinct proof that the decrypted message has been obtained correctly from its ciphertext.

The **DecryptionOracle.Decrypt** operation can be triggered by any user of the COMMON, yet it is supposed to be successful only during the **reveal** phase, and when called on suitable data.

5. **DecryptionOracle.Verify** – verifies the proof generated by **DecryptionOracle.Decrypt**. This is used in the **ORDER-BOOK** contract, when the aggregated plaintext amounts are revealed.

### 9.2 Decryption Oracle Instantiation Pattern

We provide three instantiations of the **DecryptionOracle**:

1. using Multi-Party Computation, specifically, using ElGamal threshold decryption, see Section 9.3
2. using a single party and integrating ElGamal encryption directly with the [Gro16b](#) proving system (SAVER protocol [LCKO19](#)), see Section 9.4
3. using Trusted Execution Environments, see Section 9.5

All these instantiations are based on a common pattern (see Section 9.2.1), which in turn relies on a publicly verifiable encryption scheme, as defined below.

**Definition 9.1** (Publicly verifiable threshold encryption scheme). A threshold encryption scheme $E$ is a set of seven algorithms

- **Setup**$(\lambda)$: that takes a security parameter $1^\lambda$ and outputs public parameter $pp$. These parameters determine, e.g., the maximal value of the plaintext $pp.MAXENC$, threshold of parties able to decrypt a message, number of all parties, group structure of the encryption scheme, etc.

- **KGen**(pp) → $(pk; sk_1, \ldots, sk_n)$: takes public parameters $pp$ and, if defined, the threshold $pp.thr$ and the number of parties $pp.prt$. If these are not defined it sets them both to 1 and outputs a public key $pk$, secret key shares $sk_1, \ldots, sk_{pp.prt}$ and verification key $vk$.

- **Encode**(pp, $m$): that takes a message $m \in F_p$ and outputs a vector $(m_1, \ldots, m_l)$ such that $m = \sum_{i=1}^l m_i \cdot pp.MAXENC^{i-1}$.

- **Decode**(pp, $m_1, \ldots, m_l$): that takes a vector $(m_1, \ldots, m_l)$ and outputs $m = \sum_{i=1}^l m_i \cdot pp.MAXENC^{i-1}$.
Enc(pp, pk; m; r) \rightarrow c: that takes a public key pk, message \( m = (m_1, \ldots, m_l) \) and randomness \( r \) and outputs a ciphertext \( c = (c_1, \ldots, c_l) \).

Dec(pp, sk_{i_1}, \ldots, sk_{ipthr}, pk, vk; c) \rightarrow m, \text{ proof}: that takes the pp.thr secret key shares \( sk_{i_1}, \ldots, sk_{ipthr}, \) a valid ciphertext \( c \), and outputs the plaintext \( m \) if \( m_i \leq \text{pp.MAXDEC} \) for all \( i \in [l] \), and outputs “Error” otherwise. Additionally, it outputs a proof \( \text{proof} \) that \( m \) has been obtained by correctly following the decryption procedure on \( c \).

V(pp, pk, vk; c, m; \pi) \text{ that takes a public ciphertext, plaintext } m, \text{ and proof } \text{proof} \text{ of the decryption’s correctness and outputs } 1 \text{ if and only if the proof is acceptable.}

Add(pp, c, c') \rightarrow c'' \text{ that takes two ciphertexts:}

\[ c = \text{Enc}(pp, pk; (m_1, \ldots, m_l)), \quad c' = \text{Enc}(pp, pk; (m'_1, \ldots, m'_l)) \]

such that \( m_i + m'_i < \text{pp.MAXDEC} \), for \( i = 1, \ldots, l \), and returns \( c'' = c + c' \) such that

\[ \text{Decode}(pp, \text{Dec}(pp, sk_{i_1}, \ldots, sk_{ipthr}, pk, vk; c + c')) = \text{Decode}(pp, \text{Dec}(pp, sk_{i_1}, \ldots, sk_{ipthr}, pk, vk; c)) + \text{Decode}(\text{Dec}(pp, sk_{i_1}, \ldots, sk_{ipthr}, pk, vk; c')). \]

We note that the concrete meaning of "+" depends on the encryption scheme.

Note that setting \( n = 1 \) and \( \text{pp.thr} = 1 \) we can remove the “threshold” property of the scheme. In what follows, we sometimes do so without mentioning it.

Further, we remark that our concrete instantiation based on “threshold ElGamal” [CGS97] does not fully adhere to the above formalism (see Section 37). In particular, the decryption procedure has a slightly different interface to accomodate for asynchronous communication between the decrypting parties.

9.2.1 Designing the Decryption Oracle

In Figure 9.2 we describe a general pattern for instantiating the Decryption Oracle given an encryption scheme \( E \) as in Definition 9.1. Later, we will specify \( E \) to be variants of the ElGamal “in-the-exponent” encryption scheme. Precisely, for the single-party instantiation, we use an enhanced ElGamal, as proposed in [LCKO19]. For the committee instantiation, we also use a flavor of ElGamal proposed in [CGS97].

While ElGamal “in-the-exponent”, in its basic form, is additively homomorphic, it does not allow for easy decryption of arbitrarily large numbers. This comes from the fact that for an encryption of \( m \in \mathbb{F}_{\text{scalar}} \), the decryption procedure yields the element \( m \cdot G \) and the last step boils down to solving the discrete log problem. Note that this is possible only with the guarantee that \( m \) is in some small enough range. In the schemes presented in Figs. 39 to 41, we call a function BreakDlog which takes group description from public parameters pp, generator G, and a group element H. Whenever computationally feasible, the algorithm returns a scalar x, such that \( x \cdot G = H \).
Public Call: DecryptionOracle.KGen

Input: $1^\lambda$

1. $pp \leftarrow E.\text{Setup}(1^\lambda)$
2. $(sk) \leftarrow (sk_1, \ldots, sk_{pp,pt}) : \text{SecretKey} \leftarrow E.\text{KGen}(pp)$
3. Return $(pp, sk, pk, vk)$

Public Call: DecryptionOracle.Encrypt

Input: $pk : \text{EncPKey}, m : \text{Message}, r : \text{Randomness}$

1. Assert $m \in [0, \text{MAXSUPPLY}]$
2. Set $(m_1, \ldots, m_l) \leftarrow E.\text{Encode}(pp, m)$
3. Return $c \leftarrow E.\text{Enc}(pp, pk, (m_1, \ldots, m_l), r)$

Public Call: DecryptionOracle.Add

Input: $c : \text{AHCipherText}, c' : \text{AHCipherText}$

1. Return $c'' \leftarrow E.\text{Add}(pp, pk, c, c')$

Public Call: DecryptionOracle.Decrypt

Input: $\emptyset$

1. Assert $\text{ORDER-BOOK.currentPhase} = \text{reveal}$
2. aggregate $\leftarrow \emptyset$
3. zkAggregate $\leftarrow \emptyset$
4. For each $(\text{pair}, \text{encAmount}) \in \text{ORDER-BOOK.encAggregate}$
   
   (a) $(m, \pi) \leftarrow E.\text{Dec}(pp, (sk_{i_1}, \ldots, sk_{i_{pp,thr}}), pk, vk, \text{encAmount})$
   (b) amount $\leftarrow E.\text{Decode}(pp, m)$
   // We have amount $= \sum_{i=1}^l m_i \cdot \text{MAXENC}^{i-1}$
   (c) aggregate.Add((pair, amount))
   (d) zkAggregate.Add($\pi$)
5. Return(aggregate, zkAggregate)

Public Call: DecryptionOracle.Verify

Input: $\pi : \text{ZKProof}; c : \text{AHCipherText}, m : \text{Message}$

Return $E.\text{V}(pp, pk, vk; \pi; c, m)$

Figure 38: General pattern for instantiating the DecryptionOracle based on the publicly verifiable threshold encryption scheme $E$. 
To make sure that plaintexts are efficiently decryptable we leverage the encoding procedure $\text{Enc.Add}$ described in Definition 9.1. Recall that such encoding consists of dividing the plaintext $m$ into chunks $m_i$, each in the interval $[0, \text{MAXENC})$, so that $m = \sum_{i=0}^{t} m_i \text{MAXENC}^i$, for some fixed $l$. Then, we use “ElGamal-in-the-exponent” to encrypt each chunk $m_i$ separately. Ciphertexts are thus tuples of ElGamal ciphertexts, which can then be decrypted component-wise. Finally, the original plaintext can be recovered by using the decoding algorithm from Definition 9.1.

This, however, requires some care when handling the additive homomorphic properties of the scheme. Precisely, the scheme is additively homomorphic from an “end-to-end” perspective, in the sense that, for

$$c = \text{Enc.Enc}(pp, pk, \text{E.Encode}(pp, m); r), \quad c' = \text{E.Enc}(pp, pk, \text{E.Encode}(pp, m'); r')$$

we have (as long as the components of encoded plaintexts never go beyond the bound $\text{MAXDEC}$)

$$\text{E.Decode}(pp, \text{E.Dec}(pp, sk, c + c')) = \text{E.Decode}(pp, \text{E.Dec}(pp, sk, c)) + \text{E.Decode}(pp, \text{E.Dec}(pp, sk, c'))$$

where $sk$ here denotes a tuple of secret key shares $sk_1, \ldots, sk_{\text{pp, thr}}$. However, in general it is not true that $\text{E.Dec}(pp, sk, c + c') = \text{E.Dec}(pp, sk, c) + \text{E.Dec}(pp, sk, c')$ that this equality does not necessarily hold if one removes the decoding part.

Another delicate point is that, since we perform additively homomorphic operations on encoded plaintexts, the values $m_i$ in the encoded plaintexts can, in principle, grow arbitrarily, preventing the decryption algorithm from being able to decrypt a ciphertext (due to the underlying plaintext having too large components). Fortunately, in COMMON we can get a concrete bound on the number of homomorphic operations so that no $m_i$ in a ciphertext has value beyond a constant $\text{MAXDEC}$ (being an appropriate constant which allows for solving the discrete logarithm, think of $\text{MAXDEC} = 10^{12}$).

### 9.3 Instatiation using Threshold ElGamal Encryption

We instantiate the decryption oracle by following the scheme in Section 9.2, and using Threshold Elgamal Encryption as the encryption protocol $\text{E}$. For this instantiation we assume there is a committee of $\text{prt}$ (potentially mutually distrusting) users $P_1, \ldots, P_{\text{prt}}$. When requested, the committee jointly decrypts a given ciphertext. To enforce joint decryption, we use a threshold encryption scheme that ensures that the batch is decrypted only if $\text{thr}$ parties from the committee collaborate. Moreover, any set of cardinality less than $\text{thr}$ learns nothing about the plaintext.

#### Scheme description

In Fig. 39 we provide a detailed explanation of the threshold encryption scheme we use. The proposed solution is based on [CGS97].

The key generation and decryption algorithms are performed in MPC, which requires the parties to broadcast messages to other parties and listen to other parties’ messages.
We assume the parties have established a secure broadcast channel, ensuring each broadcasted message is delivered to every party. The channel is used by calling \texttt{broadcast} (we allow to broadcast arbitrary messages, we do not discuss here the format of the messages). Similarly, each party can send a direct message to another party by calling \texttt{send} on two inputs: the recipient and the message. See Fig. 40 for the description of the decryption MPC procedure. Importantly, we keep the MPC part under the hood, leaving the decryption procedure endpoints untouched.

\textbf{Remark 9.2.} We note that in the proposed solution, we assume a trusted dealer that distributes the secret shares to the committee members. This assumption can be removed by using a \textit{publicly verifiable secret sharing scheme}, where the parties jointly compute the shares, and each party outputs a publicly verifiable proof that the shares it distributed are correct. The trusted dealer can also be substituted by a committee, e.g., the same committee that handles decryption queries.

\textbf{Public parameter instantiation, making the encryption SNARK-friendly} It is important to specify how the public parameters of the encryption scheme are chosen so as to ensure the scheme’s SNARK-friendliness. Indeed, neglecting careful picking of the parameters may cause major inefficiencies. This is especially the case when a party needs to make zero-knowledge statements about the plaintext hidden in a publicly known ciphertext, which is the case of Common.

Let \texttt{ZKP} be a zkSNARK for a relation \( R \) defined over a field \( \mathbb{F} \). That is, the circuit \( C \) that describes \( R \) is also over \( \mathbb{F} \). In order to support efficient encryption inside of SNARK relations, such as in \( R_{\text{NewOrder}} \), we need to represent \texttt{DecryptionOracle.Encrypt} as an arithmetic circuit over \( \mathbb{F} \). To this end we need a smart choice of the elliptic curve \( \mathcal{E} \) over which we use ElGamal encryption.

Recall that there are essentially two fields associated with an elliptic curve \( \mathcal{E} \), the base field \( \mathbb{F}_{\text{Base}} \) and the scalar field \( \mathbb{F}_{\text{Scalar}} \). The elements of \( \mathcal{E} \) are really tuples (pairs or triples, depending on the representation) of elements from \( \mathbb{F}_{\text{Base}} \), and the addition on \( \mathcal{E} \) can be represented efficiently as arithmetic expressions over \( \mathbb{F}_{\text{Base}} \). The scalar field \( \mathbb{F}_{\text{Scalar}} \) on the other hand can be thought of as a prime field of characteristic \(|\mathcal{E}|\) (although it is not always like this) and typically is of roughly the same size as \(|\mathbb{F}_{\text{Base}}|\). A very natural choice for \( \mathcal{E} \) is then one that satisfies the following conditions:

- the base field \( \mathbb{F}_{\text{Base}} \) of \( \mathcal{E} \) is equal to \( \mathbb{F} \),
- the Diffie-Hellman problem is hard on \( \mathcal{E} \) – so that ElGamal is safe.

For our choice of \( \mathcal{E} \) that we made in Section 2 one can indeed find such a suitable curve \( \mathcal{E} \). From now on we fix such a curve \( \mathcal{E} \) and fix its generator \( G \in \mathcal{E} \). We note however that the scalar field \( \mathbb{F}_{\text{Scalar}} \) of \( \mathcal{E} \) is not equal to \( \mathbb{F} \) (unless we have an elliptic curve cycle of length 2) but that is not an obstacle, because in the encryption procedure one just computes \( m \cdot G \) where \( m \) is a small integer – this can be easily done by including the bit decomposition of \( m \) as a hint in the witness and using exponentiation by squaring.

Differently than the trusted-party instantiation, committee instantiation has additional parameters set, namely \texttt{pp.thr}, which determines the threshold of the encryption
E.Setup($1^λ$)

1. Set $pp.$MAXLENGTH $\leftarrow \log_2 \text{MAXENC}$ \hfill \text{(MAXENC is a power of 2)}
2. Set $pp.$maxNumber $\leftarrow l$
3. Set $pp.$group $\leftarrow \mathcal{E}$
4. Set $pp.$group[0].generator $\leftarrow G$
5. Set $pp.$thr $\leftarrow t$
6. Set $pp.$prt $\leftarrow n$
7. Set $pp.$PublicKey $\leftarrow \mathcal{E}$
8. Set $pp.$SecretKey $\leftarrow \mathbb{F}_{\text{scalar}}$
9. Set $pp.$VerificationKey $\leftarrow \perp$
10. Set $pp.$Message $\leftarrow [0, \text{MAXSUPPLY}]$
11. Set $pp.$AHCipherText $\leftarrow \mathcal{E}^{2l}$

Return $pp$

E.KGen($pp$)

1. For $j = 1, \ldots, t$
   (a) $sk_j \leftarrow \mathbb{F}_{\text{scalar}}$
   (b) $\text{send}(P_j, sk_j)$
   (c) $\ell_j(X) \leftarrow \prod_{i \in [1, \ldots, l] \setminus \{j\}} \frac{X-i}{j-i}$
2. $f(X) \leftarrow \sum_{i=1}^l \ell_i(X) \cdot sk_i$
3. For $j = t+1, \ldots, n$
   (a) $sk_j \leftarrow f(j)$
   (b) $\text{send}(P_j, sk_j)$

4. $pk \leftarrow G \cdot \sum_{i=1}^l \ell_i(0) \cdot sk_i$

Return $sk, pk, vk \leftarrow \perp$

E.Enc($pp, pk, m = (m_1, \ldots, m_l); r = (r_1, \ldots, r_l)$)

1. For $i = 1, \ldots, l$
   $c_i \leftarrow (r_i \cdot G, m_i \cdot G + r_i \cdot pk)$
2. Return $c = (c_1, \ldots, c_l)$

E.Dec($pp, (sk_1, \ldots, sk_n), pk, vk, c$)

See Fig. [40] for details.

E.AuxV($pp, pk, vk; c, \pi = (a, b, h, z, o_v)$)

// Auxiliary verification procedure used in E.V and in E.Decrypt
1. Parse $(c^1, c^2) \leftarrow c$
2. Return $(z \cdot G = a + h \cdot pk_j) \land (z \cdot c^1 = b + h \cdot o_v)$

E.V($pp, pk, vk; c, m, \pi$)

1. Parse $(\pi_j^l)_{j,v \in [\text{thr}], j \neq v} \leftarrow \pi$
2. Parse $(c_1, \ldots, c_{\text{thr}}) \leftarrow c$
3. Return $E.$AuxV($pp, pk, vk; c_j, \pi_j^v$) for all $j, l \in [\text{thr}], j \neq l$

// Note $m$ is not used in this procedure, though we include it as input due to our formalization in Definition 9.1

E.Add($c, c'$)

1. Assert $|c| = |c'|$
2. For $i = 1, \ldots, |c|$ do $c''_i \leftarrow c_i + c'_i$

Return $c''$

Figure 39: Publicly verifiable threshold encryption scheme based on ElGamal.
E. Dec

On input \((pp, sk_v, pk, vk, c)\):

1. Parse \((c_1, \ldots, c_l) \leftarrow c\)
2. For \(j = 1, \ldots, l\),
   (a) \(\text{counter}_j \leftarrow 0\)
   (b) \((\text{aux}_j, \pi_j) \leftarrow (\perp, \perp)\)
   (c) \(\text{isDecrypted} \leftarrow \text{False}\)
   (d) Parse \((c^1_j, c^2_j) \leftarrow c_j\),
   (e) \(M^v_j \leftarrow sk_v \cdot c^1_j\)

   / In the following steps \(P_v\) generates a proof \(\pi^l_v\) of correct decryption
   (f) Pick a random \(r'_j \leftarrow \mathbb{F}_{\text{scalar}}\)
   (g) Compute \((a_j, b_j) \leftarrow (r'_j \cdot G, r'_j \cdot c^1_j)\)
   (h) Compute \(h_j \leftarrow \text{Hash}(pp, (M^v_j, c^1_j), a_j, b_j)\)
   (i) Compute \(z_j = r'_j + sk_v \cdot h_v\)
   (j) \(\pi^l_v \leftarrow (a_j, b_j, h_j, z_j, M^v_j)\)
   (k) broadcast(VerifyPartialDec, \((c_j, j; \pi^l_v)\))

On input \((\text{VerifyPartialDec}, (c_j, j; \pi^l_v'))\) from \(P_v\):

1. If \(E.\text{AuxV}(pp, pk, vk; c_j, \pi^l_v')\), then
   \(\text{counter}_j[\nu'] \leftarrow 1\)
   \((\text{aux}_j, \pi_j) \leftarrow ((\text{aux}_j || (c_j, j), (\pi_j || \pi^l_v'))).\)
2. If \(\sum_{k=1}^\nu \text{counter}_j[k] = \text{thr}\) then
   (a) \(\Lambda_j \leftarrow \{k : \text{counter}_j[k] = 1\}\)
   (b) \(M_j \leftarrow c^2_j - \sum_{\Lambda_j} M^v_j, k \cdot \prod_{i \in \Lambda_j \setminus k} t_i \quad \text{// } m_j \cdot G \text{ reconstruction}\)
   (c) \(m_j \leftarrow \text{BreakDlog}(pp, G, M_j)\)
   (d) If \(m_j \neq \perp\) then \(\text{isDecrypted}[j] \leftarrow \text{True}\)

If \(\forall j \in [1, \ldots, l], \text{isDecrypted}[j]\) then return \(m \leftarrow (m_1, \ldots, m_l), \text{aux} \leftarrow (\text{aux}_1, \ldots, \text{aux}_l), \pi \leftarrow (\pi_1, \ldots, \pi_l)\)

Figure 40: Extended view of the decryption algorithm from the perspective of party \(P_v\)
scheme, and \( \text{pp.prt} \) which corresponds to the number of all parties that can participate in decryption.

### 9.4 Instatiation with a Single Party using SAVER

Another instantiation we propose relies on a trusted third party that keeps the decryption key. To ensure that the oracle cannot cheat on decryption, we require it to provide a proof of the decryption’s correctness.

**Public parameter instantiation, using SAVER**  
For the trusted third-party instantiation, we use an enhanced ElGamal encryption scheme, as proposed by Lee et al. in [LCKO19], see Fig. 41. [LCKO19] allows for easy verification of the decrypted messages and efficient proving facts about the plaintext by a smart modification of the Groth16 zkSNARK [Gro16a]. Since the Groth16 proof system requires bilinear pairings, we set \( \text{pp.prt} \), to include descriptions of three groups \( G_1, G_2, G_T \). We denote by \( e : G_1 \times G_2 \rightarrow G_T \) a pairing. We require that all these groups have order \( \text{SFIELD} \), \( G_1, G_2 \) are defined over elliptic curves. For the sake of concreteness, we denote by \( G \) a generator of \( G_1 \) and by \( H \) a generator of \( G_2 \). We also recall that according to our convention in Section 2 the scalar field of curves \( G_1 \) and \( G_2 \) is denoted by \( F \).

Lee at al. propose the following approach. Let \( R \) be a relation defined over \( F \). We denote by \( C \) the arithmetic circuit that represents \( R \). We note that \( C \) is defined over \( F \).

The key idea of [LCKO19] is to observe that the ElGamal encryption scheme considered over \( G_1 \), i.e., such that the plaintext messages are in \( F \) and the ciphertexts are in \( G_1 \) is very much compatible with the proof-generation machinery in [Gro16b]. What this allows to do is to integrate this encryption scheme “natively” into the Groth16 proof-system. More specifically, SAVER is an extension of Groth16 that allows to include as part of the statement not only elements of \( F \), but also ciphertexts (in \( G_1 \)) that are ElGamal encryptions of arbitrary witnesses. Exactly what we need in the relation \( R_{\text{NewOrder}} \).

**Homomorphic additivity of SAVER’s ElGamal**  
For a public key

\[
\text{pk} = (X_0, X_1, X_2, \ldots, X_l, Y_1, Y_2, \ldots, Y_l, Z_0, Z_1, Z_2, \ldots, Z_l, W_1, W_2, \ldots, W_l)
\]

the encryption of a message \( m = (m_1, \ldots, m_l) \) under randomness \( r \) equals \( \text{Enc}(\text{pp, pk, m, r}) = (r \cdot X_0, (r \cdot X_1 + m_1 \cdot G_1), \ldots) \). Similarly, an encryption of \( m' = (m'_1, \ldots, m'_l) \) under randomness \( r' \) and key \( \text{pk} \) equals \( \text{Enc}(\text{pp, pk, m', r'}) = (r' \cdot X_0, (r' \cdot X_1 + m'_1 \cdot G_1), \ldots) \). Adding these two ciphertexts gives us an encryption (one of possibly many) of \( m + m' = (m_1 + m'_1, \ldots, m_l + m'_l) \) under randomness \( r + r' \) and public key \( \text{pk} \). That is,

\[
\text{Decode}(\text{Dec}\text{(pp, sk, Enc(pp, pk, m, r)}) + \text{Dec}(\text{pp, sk, Enc(pp, pk, m', r'))}) =
\]

\[
= \text{Decode}(\text{Dec}(\text{pp, sk, Enc(pp, pk, m + m', r + r'))}).
\]
Figure 41: Publicly verifiable encryption scheme based on the SAVER protocol.
9.5 Trusted Execution Environment Instantiation

The third instantiation is by using a Trusted Execution Environment (TEE). In that case, the Decryption Oracle is simply some secure hardware that contains the decryption key and:

- uses the key only to decrypt whatever is requested by the ORDER-BOOK contract,
- does not allow to extract the decryption key by any party.

The main challenge when using this type of instantiation is to ensure the first condition above is satisfied (with the second condition being guaranteed by the TEE manufacturer). We propose the following two ideas – their technical viability might depend on the concrete type of TEE used:

1. One could let the TEE run a light client of the blockchain. This way, the TEE operator can prove to the TEE, using Merkle proofs, what the current round and phase is and what is the content of \texttt{encAggregate} in the ORDER-BOOK contract. This way the TEE could validate the "decrypt conditions" are satisfied and proceed only if a correct proof is given.

2. Upon each decryption, the TEE could be instructed to sign a nonce signifying how many decryptions did the TEE perform so far. This signed nonce would need to be posted on chain (along the decryption) by the TEE operator. This way, the first event when the TEE was used to decrypt something out of order would be immediately visible. Note that this does not really defend against an adversary that wants to just reveal the whole history and is fine with being detected. However, we can disincentivize the operator from doing so by asking them to stake some funds that are potentially slashed on misbehavior.

10 Extensions and Practical Considerations

10.1 Recovering from Decryption Oracle Malfunction

An important property of \texttt{COMMON} is that the security of users’ funds does not require assumptions on the Decryption Oracle. The basic version of the exchange presented in the previous section is already safe against stealing funds by the Decryption Oracle. Indeed, in every round the Decryption Oracle makes only one decision: either reveal the plaintext aggregated values or not, and it doesn’t have any impact on where users’ funds go. Note that the oracle could also reveal incorrect decryptions – but this is equivalent to not revealing anything at all, because the decryptions are verified, and thus incorrect decryptions would simply be ignored.

However, already from this decision (reveal or not) there comes some significant power – by not revealing, the Decryption Oracle could cause the round to get stuck on the \texttt{reveal} phase. This would cause the funds that went into this round’s order batch to be frozen. Depending on the concrete instantiation of the Decryption Oracle and the underlying assumptions the likelihood of this event might vary, but taking hardware and
software failures into account it’s not possible to rule it out entirely. That is why ideally there should be a mechanism that prevents fund freeze when the Decryption Oracle stops responding.

Fortunately, there exists a straightforward modification to COMMON that allows to deal with this issue. Indeed it is enough to add the possibility of cancelling the round when the \textit{reveal} is taking too long. Roughly the wait it would work would be the following:

- If the \textit{reveal} takes too long, i.e. the call ORDER-BOOK.RevealTradeValues is not successfully called for a given number of blocks, then the possibility to make a new call ORDER-BOOK.CancelRound is unlocked.
- When ORDER-BOOK.CancelRound is called, the round is considered void. In particular a series of \textit{TradeEvent}s is emitted with \textit{amountTraded} = 0 for each.
- The users can cancel their orders at any time (using ORDER-BOOK.CancelOrder. This way they can avoid their orders to be included in any other batches, and just call ORDER-BOOK.UpdateOrder and ORDER-BOOK.ClaimCancelled to withdraw the funds.

This modification, while simple, makes COMMON completely safe against any faults of the Decryption Oracle.

\section*{10.2 Compliance and Avoiding Bad Actors}

When building privacy systems on blockchains an important problem to tackle is that of bad actors who try to privacy systems to launder funds obtain by illicit activities. Multiple solutions has been proposed for this problem (see for instance the recent work \cite{BIN+23}). This topic has been purposely left out of scope of this paper since it is independent of the problem we are solving here (private DEX). Indeed, most of the aforementioned solutions can be integrated at the shielded pool layer, and do not require any changes to the COMMON design.

\section*{10.3 Correlations Based on Transaction Origin}

It is worth mentioning that when using a shielded pool one needs to be quite careful with regard to sending transaction. Indeed, consider the following antipattern:

\begin{enumerate}
  \item User creates an account $A$ on the blockchain.
  \item The user then deposits 1 ETH of its funds from account $A$ to the shielded pool.
  \item After enough time, the user withdraws 1 ETH from the shielded pool to the account $A$.
\end{enumerate}

Note that this way the user gained no privacy, because the actions of depositing and withdrawing are linked in an obvious way: both originate from the same account. Indeed for unlinkability it is absolutely crucial to perform the withdraw to a different account than $A$.  

77
Whereas the above suggestion sounds simple and obvious, in practice, it is not so easy to follow when the shielded pool has an interface exactly as specified in Section 6. The reason is that in order to initiate a withdraw to a different account $B$ than the deposit, the user needs native coins to pay the gas fee. This however requires to fund the account $B$ with coins for gas. However, if the user tops up $B$ from $A$, this again creates a clear link between these accounts and hence between deposit and withdrawal.

There are a few well known solutions to this problem. In a practical deployment of COMMON it’s important to implement one of these to circumvent the discussed issue. One of the solutions is to implement a way to fund fresh accounts on demand to cover gas cost. This can be achieved by the means of a semi-trusted party which sells "tickets" for creating fresh accounts using blind signatures. These tickets are then redeemed by communicating with the party off-chain. If blind signatures are used to create these tickets then the party cannot link any particular ticket to who bought it.

Another potential solution is to modify the ORDER-BOOK WithdrawTokens call to specify a withdraw address that is separate from the caller. This allows to have a "relayer" role that is responsible for sending transactions on chain that withdraw tokens to possibly empty accounts. This is often accompanied also with an incentive layer for relayers, where part of the withdrawn funds go to the transaction sender.

### 10.4 Correlations Based on Token Amounts

One particularly successful technique of deanonymizing users of shielded pools is the analysis of deposited and withdrawn amounts of particular tokens. Here is the basic idea: if there was a deposit for a very specific amount, say 3.417 tokens, and after a while a withdraw for that exact amount is performed, then there is a good chance the deposit and withdraw were made by the same user, hence there is a high probability link between these transactions. Even if the withdraw amount is not the same as the deposit, but it is something like 2.417 there is still a good probability that it’s the same user, who just withdrew 1 token first, and the rest later.

There are a few possible mitigations for this issue: one that is employed in many different protocols is to limit the possible token amounts deposited and withdrawn in the system to just a few possibilities (say 1, 5, 25 ETH). This makes the analysis based on token amounts impossible.

In the case of a system like COMMON the above mitigation is not so straightforward to apply – even if we force the users to deposit amounts from a short prespecified list, they will end up holding different amounts because of the trades they have performed in the DEX. One can still apply the same idea by putting constraints on withdraw amounts (and not on deposit), for instance: the users can withdraw amounts that are powers of 2. This way, a user who wants to withdraw 67 AZERO would need to withdraw 64, then 2 and then 1. Note that this increases the costs (transaction fees), is less convenient, and also barely increases the privacy in case the user performs the transactions one by one.

An alternative to the above, instead of enforcing rules on deposit and withdrawal at the protocol level, is to allow the user interface (wallet software) to take care of that. This solution is way more flexible and allows the user to configure a privacy vs
convenience/cost tradeoff. The basic idea is as follows: whenever the user interacts with
the shielded pool, it’s notes are intentionally broken into pieces of random denominations.
Also, when withdrawing, the user would do that in pieces, ideally not at the same time,
but leaving random time intervals between withdrawals. How much of this "obfuscation"
happens should be configurable and also can depend on the current state and traffic in
the pool.

10.5 Adding Noise to Hide Order Direction

Although in our protocol the users reveal the direction of their orders when they send
their transactions to the ORDER-BOOK, the fact that the order values are hidden allows for
fake orders that encrypt “zero” to be created to disrupt information collection about the
directions. In that case the encryption scheme should be IND-CPA secure in order to
prevent an attacker from distinguishing the orders that encrypt “0” from the other orders.
For example, users could create simultaneously two orders with opposite directions, one
real and one “fake” that encrypts “0”. The idea of using “dummy” outputs with zero
value has been used also in [HBHW22] to hide the number of inputs and outputs of a
transaction.

A Precision of Fixed Point

In Section 2.2 we defined addition and multiplication operations between values of type
FixedPoint. These can be thought of as “approximately modelling operations between
rational numbers”. The lemma below describes the error incurred by these operations,
with respect to the rational operations they model.

Lemma A.1 (Small errors). Let $x_1, x_2, x_3 \in \mathbb{Q}$ be three rational numbers such that
$x_i = y_i/M$ for some fixed point numbers $y_1, y_2, y_3 \in B$. Then, if $y_1 + y_2 < B$,

$$y_1 + y_2 = y_3 \quad \text{if and only if} \quad x_1 + x_2 = x_3.$$  

Similarly, it also holds that, if $y_1 y_2 < B$,

$$y_1 \cdot y_2 = y_3 \quad \text{if and only if} \quad 0 \leq x_1 \cdot x_2 - x_3 < \frac{1}{M}.$$  

In both equations, the operations on the left are between values of type FixedPoint, and
the operations on the right are over the rationals numbers, as defined in Section 2.2.

Proof. The first statement concerning addition operations is clear. To prove the state-
ment concerning rational multiplication and the operation $\cdot$, observe (all operations below
are rational number operations)

$$0 \leq x_1 x_2 = \frac{y_1}{M} \frac{y_2}{M} = \left( \frac{\lfloor y_1 y_2 \rfloor}{M} \frac{M + r}{M^2} \right) = \frac{\lfloor y_1 y_2 \rfloor}{M} + \frac{r}{M^2} < \frac{\lfloor y_1 y_2 \rfloor}{M} + \frac{1}{M}$$  

(1)
where $0 \leq r < M$. Hence, if $y_1 \cdot y_2 = \lfloor y_1 y_2 / M \rfloor = y_3$, then $0 \leq x_1 x_2 - x_3 < 1/M$.

Conversely, if $0 \leq x_1 x_2 - x_3 < 1/M$, then, using (1),

$$0 \leq x_1 x_2 - x_3 = \left\lfloor \frac{y_1 y_2}{M} \right\rfloor + \frac{r}{M^2} - \frac{y_3}{M} < \frac{1}{M}$$

$$\iff 0 \leq \frac{y_1 y_2}{M} - y_3 + \frac{r}{M} < 1.$$  

Since both $q := \left\lfloor \frac{y_1 y_2}{M} \right\rfloor$ and $y_3$ are integers we have that either they are the same integer (as wanted), or $|q - y_3| > 1$. Since $r \geq 0$, the above inequality can only hold if $q = y_3$, as needed.

References

[lin] 1inch network. [https://1inch.io](https://1inch.io)


Theodoro Constantinides and John Cartlidge. Block auction: A general blockchain protocol for privacy-preserving and verifiable periodic double auctions. In Yang Xiang, Ziyuan Wang, Honggang Wang, and Valtteri Niemi,


[cow] Cow protocol. [https://cow.fi](https://cow.fi)


