ClockWork: An Exchange Protocol for Proofs of Non Front-Running

Dan Cline: University of Massachusetts, Amherst*
Tadge Dryja, Neha Narula: MIT DCI

*Research conducted while working at the MIT Digital Currency Initiative

Exchange Systems

Exchange Systems allow users to trade assets with each other at some exchange rate.

- We focus on **centralized** exchange systems
- These are not necessarily cryptocurrency exchanges
- Users post **orders** to trade assets
- Orders get **matched** to execute a trade of assets
- Order contents (price, amount) are visible to all

Front-Running

Front-running is when the exchange makes a decision on how to match orders based on the contents of the orders themselves.

This allows the exchange to gain some sort of profit or advantage risk-free.

- This is bad - the user took the risk, not the exchange!
- The exchange is stealing from users!

Now imagine Alice and Bob trying to trade on an exchange…

Front-Running: Insertion

The exchange can **insert** its own orders based on the contents of orders before matching:

Alice places a sell order for an asset at $5

Bob places a buy order for an asset at $10

Exchange sees this, inserts orders which buys the asset from Alice for $5 and sells it back to bob at a price of $10

- Exchange is stealing profit from Bob using insider knowledge

Front-Running: Dropping

The exchange can **drop** users' orders based on the contents of orders before matching:

Alice places buy order at $10

Bob places sell order at $9

Exchange places sell order at $10

Exchange drops Bob's order, gets to sell at $10

Front-running by insertion and dropping motivate **blind commitment**

- We do not want the exchange to see orders before committing to matching them.

Security Goals

Blind Commitment: The exchange commits to a batch without knowing any information about the orders in the batch.

A strawman design - Commit/Reveal

Let’s try to design a system to achieve this.

We don’t want the exchange to know orders before committing to matching them.

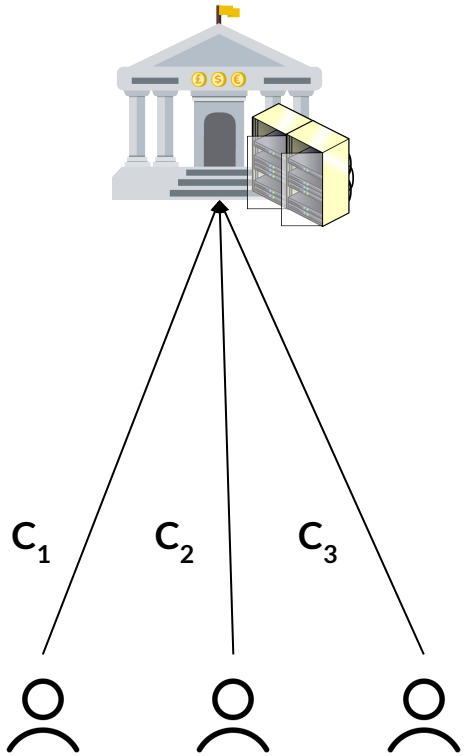
So let’s have users submit commitments to orders, and reveal them after the exchange has signed their commitment.

A strawman design - Commit/Reveal

Alice, Bob, and others would like to trade on an exchange.

They take their orders and commit to them, send them to the exchange.

$\mathbf{C_1} \leftarrow \mathbf{Commit}(O_1)$
$\mathbf{C_2} \leftarrow \mathbf{Commit}(O_2)$
$\mathbf{C_3} \leftarrow \mathbf{Commit}(O_3)$

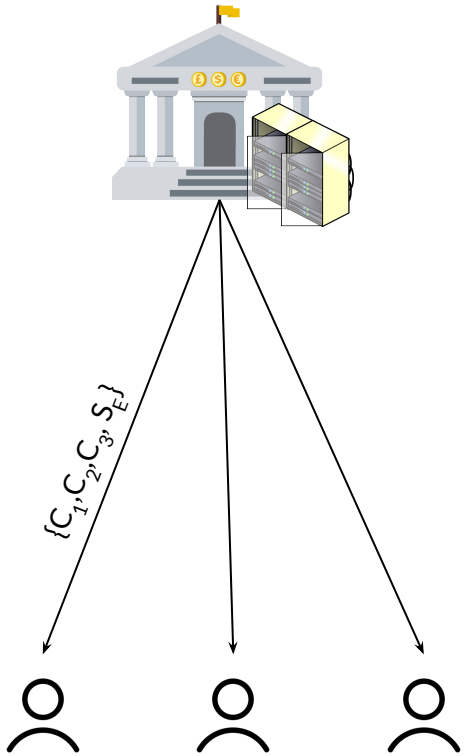


A strawman design - Commit/Reveal

Then, the exchange signs this set of orders:

$$S_E \leftarrow \text{Sign}_E(C_1, C_2, C_3)$$

And sends them to users.



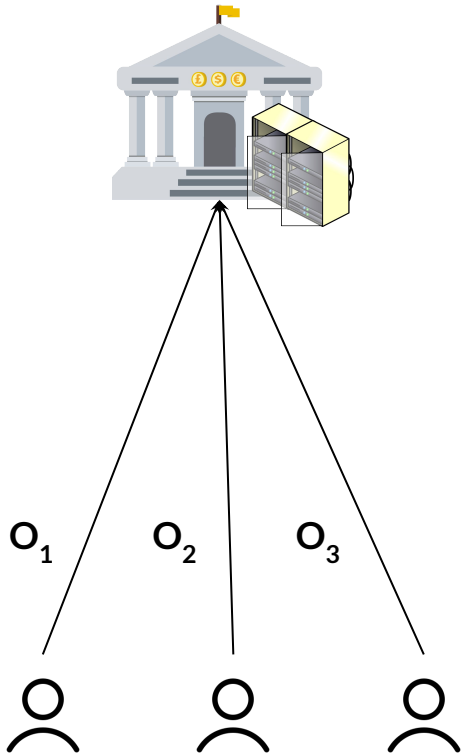
A strawman design - Commit/Reveal

Then, the exchange signs this set of orders:

$$S_E \leftarrow \text{Sign}_E(C_1, C_2, C_3)$$

And sends them to users.

Finally, the users reveal their orders, and the exchange computes the result of the matching algorithm, and executes trades.



Does Commit/Reveal solve front-running?

The exchange cannot insert orders after committing...

But what happens if not every order is revealed?

Option 1: Execute batch anyway

- The exchange can pretend it never received a users' reveal, users can't tell
- Exchange can insert many orders and drop subset once they see revealed orders

Option 2: Throw out batch

- Users can halt the system by not revealing
- Exchange can throw out un-advantageous batches after reveals

Alternative solutions

Bonded Commit/Reveal is an alternative to normal Commit/Reveal - but what does it solve?

- Incentivizes users to reveal rather than abort
- This is good - disincentivizes malicious users

Downsides:

- Ties up capital
- Still might be profitable to front-run
- Exchanges can censor reveals, making it appear that users failed to reveal and lose their bond

Security Goals

Blind Commitment: The exchange commits to a batch without knowing any information about the orders in the batch.

Binding Execution:

- All valid committed orders included in execution

Liveness:

- Invalid orders or malicious users cannot halt order execution

Can we prevent front-running?

Can users get evidence that the exchange did not front run *their* order?

The ClockWork Protocol

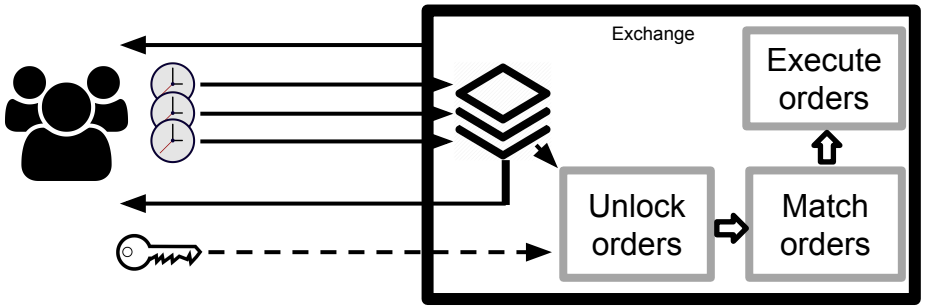
ClockWork consists of four main steps:

Setup: Exchange sends parameters to users.

Send: Users send puzzles to exchange.

Commit: Exchange commits to a set of puzzles.

Open: Exchange solves puzzles and users attest non front-running to exchange.



Before attesting, a user is convinced the exchange could not have front-run their order.

Key Insight

Timed Commitments [BN00] are a great idea for implementing an exchange protocol!

We use Timed Commitments and Timelock Puzzles [RSW96] to make sure the exchange has no way of knowing a set of orders before committing to matching them.

Timed commitments and timelock puzzles let us **guarantee** the eventual opening of puzzles.

Timed Commitments

We use timelock puzzles to achieve the same features as Timed Commitments:

(c,N,p) ← **Timelock**(m,t)

Using the puzzle trapdoor, one can unlock the message quickly:

m ← **TimeUnlockFast**(t,c,N,p)

But if there is no trapdoor, the message can still be obtained slowly (in t steps):

m ← **TimeUnlockSlow**(t,c,N)

Timelock Puzzles

Timelock puzzles take a message m and create a puzzle that cannot be solved in less than t steps:

- First, compute a modulus N=pq
- Next, compute φ(N) = (p-1)(q-1)
- Compute puzzle result b = 2^{2^t (mod φ(N))} (mod N)
- Create a symmetric key from b
- Encrypt order with AES-GCM, returning ciphertext c_i.

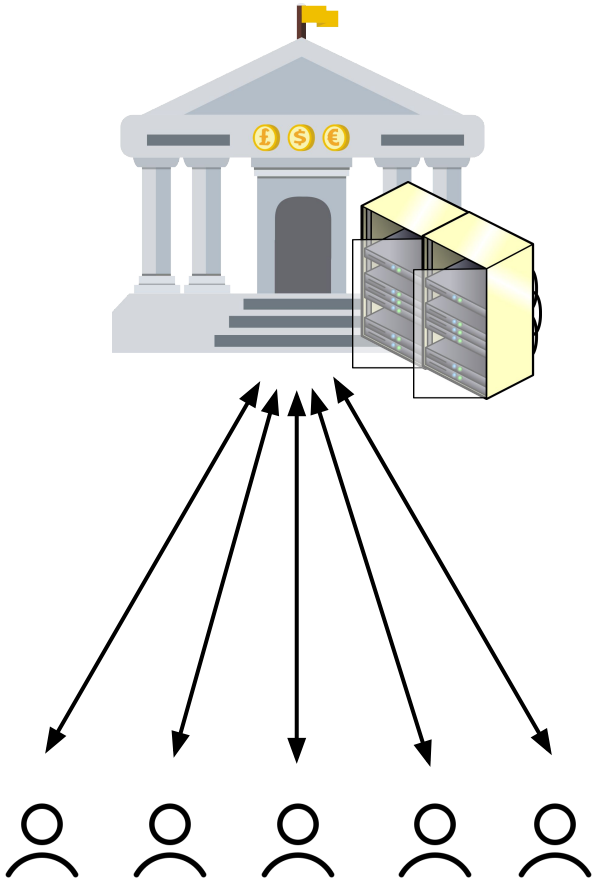
System Model

The exchange:

- Accepts incoming orders
- Orders settled based on public matching algorithm

Participants: Users $u_1, \dots, u_i, \dots, u_n$ which place orders on the exchange.

We want to constrain the exchange from carrying out front-running.



Threat Model

The Exchange wants to match *some* users' orders in order to make money, but otherwise may be malicious.

Users want to trade, but may want to gain an advantage over one another, or disrupt the trading process. We model the users as potentially malicious.

The Exchange has access to many parallel computing resources.

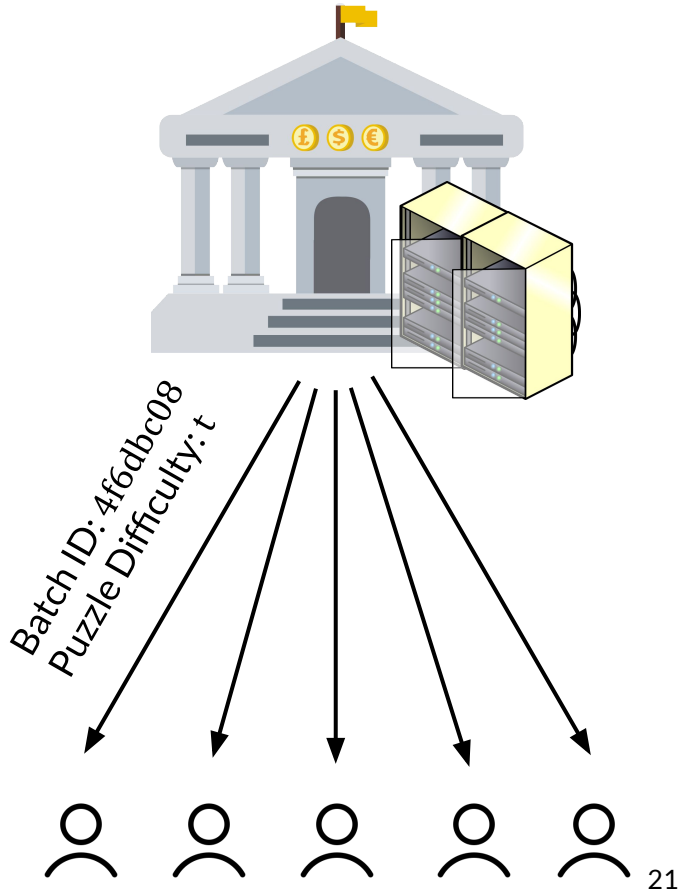
Even if all other users collude with the exchange, we want the user to have a guarantee that they were not front-run.

ClockWork Design: Setup

The exchange sends a unique batch ID and puzzle difficulty parameter t to users.

The exchange also publishes the matching algorithm M it will be using for the batch.

Users come up with a safety parameter $\Delta_i = ct$.



ClockWork Design: Send Orders

Then users encrypt their order and create a corresponding timelock puzzle:

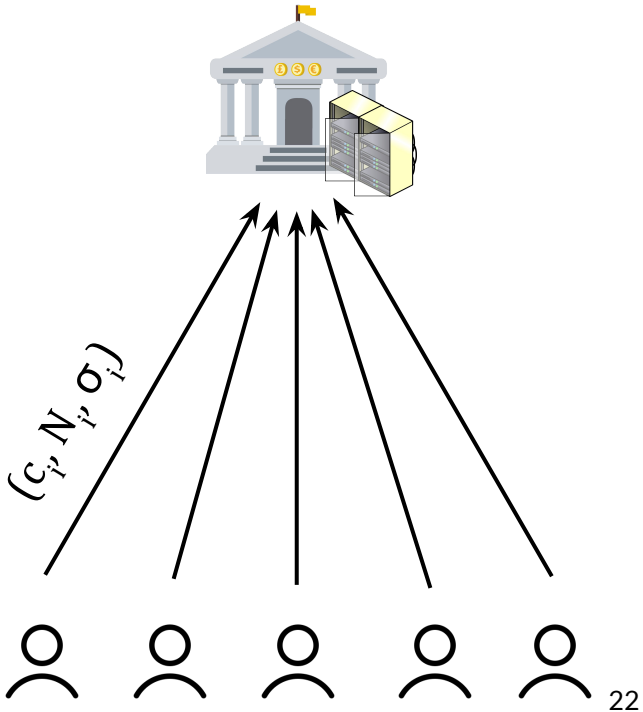
$(c_i, N_i, p_i) \leftarrow$ **Timelock**(O_i, t).

They then sign this data:

$\sigma_i \leftarrow$ **Sign**$_E((c_i, N_i))$

And sign and send (c_i, N_i, σ_i) to the exchange.

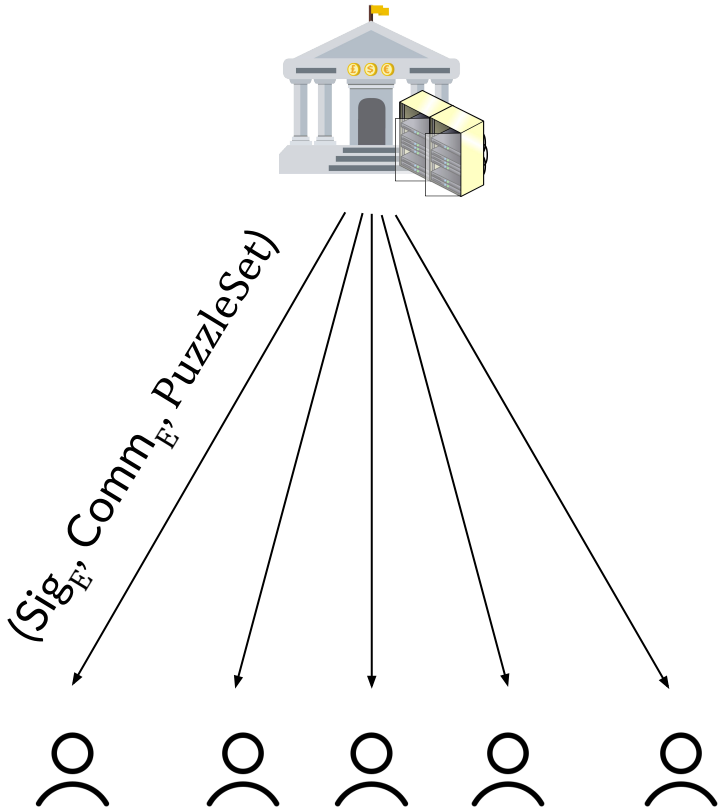
This is efficient for users - they generate the trapdoor p_i



ClockWork Design: Commit

The Exchange verifies all signatures σ_i for all i and creates a commitment to the puzzles.
Let's call the set of all (N_i, c_i, σ_i) the PuzzleSet.

The exchange signs the set
$\text{Sig}_E \leftarrow \text{Sign}(\text{PuzzleSet})$
and sends $(\text{Sig}_E, \text{PuzzleSet})$ to users.

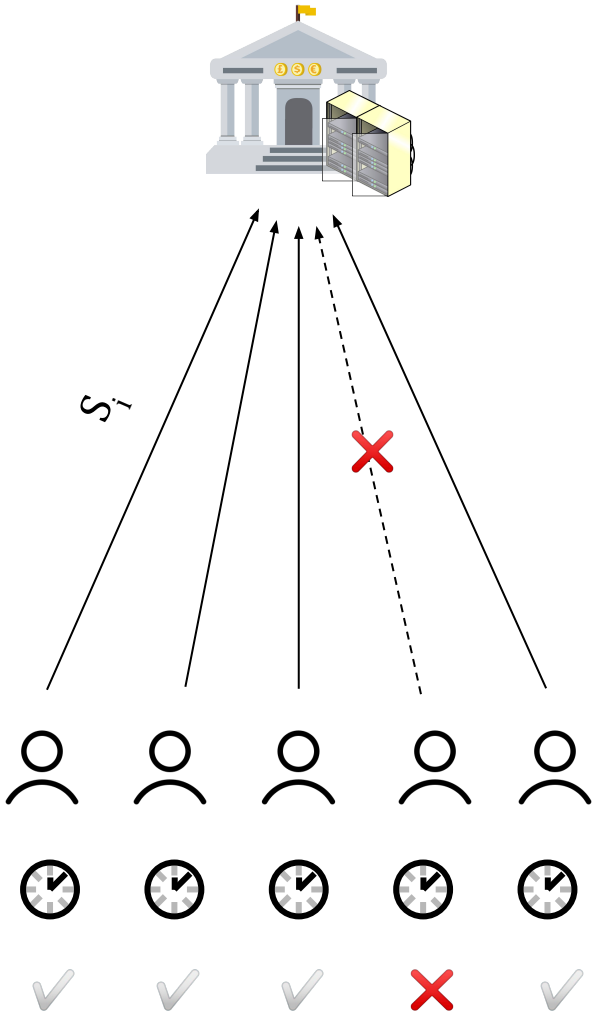


ClockWork Design: Attest

Users mark the difference in time between submitting an order and receiving the commitment.

If this difference in time $\Delta_i' > \Delta_i$ for user i, then user i does not sign. Otherwise, user i signs and sends the data received from the exchange and the trapdoor:

$$S_i \leftarrow \text{Sign}(\text{Sig}_E, \text{PuzzleSet}, p_i)$$

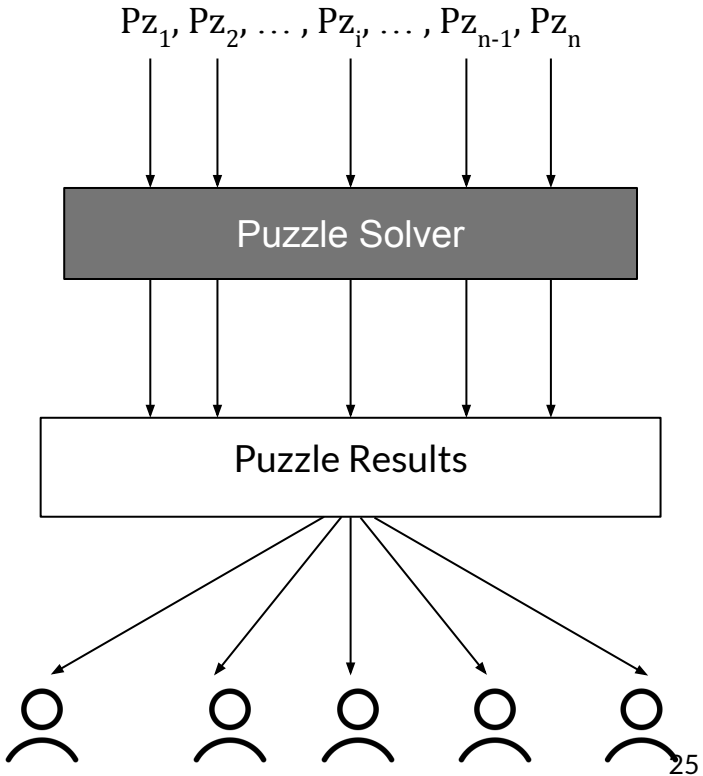


ClockWork Design: Open

Finally, the exchange solves the timelock puzzles and sends the results to users.

The exchange can solve MANY puzzles in parallel, but not ONE puzzle.

Some puzzles do not **need** to be solved because users received a commitment in time and sent trapdoor.



Implementation

Protocol implemented and benchmarked in Go:

- https://github.com/mit-dci/opencx

Future Work

- Would love to see applications of this to smart contract-based exchange!

Thank You!