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

Dan Cline
University of Massachusetts Amherst[*]

Thaddeus Dryja
MIT Media Lab

Neha Narula
MIT Media Lab

Abstract

Exchanges are critical for providing liquidity and price transparency to markets, but electronic exchanges sometimes front run their users: because the exchange is in a privileged position, it can observe incoming orders and insert its own orders or alter execution to profit, if undetected, risk-free. There are cryptographic schemes to address front-running, but they either require an assumption of non-collusion or do not definitively prevent it, and none can provide the exchange with useful evidence of good behavior: a transcript the exchange can show to an offline entity, like a potential new customer or a regulator, to prove that it is not front running.

We present ClockWork, a practical exchange protocol which gives an exchange the ability to *prove* to a user that it did not front-run their order. In ClockWork, users commit to and encrypt orders inside a timelock puzzle. By assuming a lower bound on the time it takes to solve the puzzle, we ensure that no one, including the exchange, can submit new orders or selectively drop orders after the batch is fixed, and that users cannot repudiate committed orders. Users interacting with the exchange are convinced that the exchange did not front-run, and the protocol creates a transcript between the exchange and the users that serves as evidence orders were matched correctly and has attestations from users who agree they were not front-run. We implement ClockWork and show that despite using computationally expensive timelock puzzles, it provides reasonable performance for batch auctions . This is a useful tradeoff to provide a verifiably correct exchange.

1 Introduction

Cryptocurrency exchanges enable users to purchase and sell digital assets and account for hundreds of millions of dollars in transactions every day [12]. Unfortunately, the most popular of these exchanges operate in low-regulation jurisdictions and cryptocurrency exchange operators have been suspected of stealing users' funds, wash trading, and front-running [12].

One area where we have little insight is in how often cryptocurrency exchanges are front-running their users. Typically, an exchange is responsible for collecting and ordering the bid (offer to sell) and ask (offer to buy) transactions from multiple users, and then running a matching algorithm to pair bids and asks. An exchange could manipulate the ordering or insert new orders based on the incoming stream to advantageously change the results of the match. This can lead to what is known as *front-running*, which is when a broker, exchange, or other intermediary responsible for transaction ordering manipulates that ordering for personal benefit. Front-running leads to a breakdown in market trust and integrity because some actors have access to information that others do not have when placing their orders. Though illegal, there are examples of front-running in exchanges and banks, resulting in fines of hundreds of millions of dollars [13–15]. For example, HSBC was fined $63M by the US Department of Justice and Credit Suisse was fined $135M by the New York State Department of Financial Services, both for front-running their customers. Cryptocurrency users have complained of front-running on exchanges [1, 11]. Because users have no insight into how these exchanges operate, it is impossible to determine whether or not these allegations are true.

This paper presents ClockWork, a protocol for a centralized cryptocurrency exchange that allows the exchange to *prove* to a user that it did not front-run their order. In ClockWork, users submit orders encrypted with a timelock puzzle, which is a puzzle that requires sequential work to open. ClockWork uses the timelock puzzles described in Rivest et al. [19]. The exchange then commits to a batch of orders to match *before* it has a chance to decrypt any of the orders, meaning it could not have obtained any new information from submitted orders to influence its own orders. Users sign the batch to acknowledge that the exchange has done so in a timely manner, giving the exchange a record of its correct execution. Because ClockWork uses a timelock puzzle and orders are broadcast, given enough time, the exchange can eventually reveal any order,

[*]This research was conducted while working at the MIT Digital Currency Initiative.

unlike other commit/reveal schemes where a user (or the exchange acting like a user) might choose to never reveal an order based on other revealed orders. This means that all committed orders must be matched. The transcript of messages between the exchange and users and the users' signatures serve as proof that users agree the exchange did not front-run their orders in this batch.

Because a user might not open their timelock puzzle, in the worst case the exchange has to solve a timelock puzzle for each order in a batch. This can be done in parallel and we evaluate the performance on orders of various sizes on CPUs and GPUs.

Previous work by Thorpe and Parkes [21] has aimed to address front running by matching under homomorphic encryption, so that the exchange never sees the orders in cleartext until they are matched. This requires a third-party bulletin board that does not collude with the exchange; unfortunately, no centralized exchanges that we know of are actually built to work this way in practice. Work by Khalil et al [10] also uses timelock puzzles to address front running but cannot prevent it entirely and the exchange cannot produce a transcript to show others that it has not front run.

To summarize, the contributions of this work are as follows:

- ClockWork, a system for an exchange to prove to a user that it did not front run their order in a batch auction
- A design for an offline auditor to confirm that known users were not front run
- An implementation of ClockWork and evaluation showing that the scheme is low overhead on clients but requires a core per order in the batch on exchanges, making it well-suited to GPUs.

2 Background and Related work

Exchange systems. Exchanges are composed of an *orderbook*, a *matching engine*, and a *settlement layer*. The orderbook is a set of outstanding trade requests. The matching engine runs an order matching algorithm on the orderbook to produce a set of trade executions (matched orders) and a new orderbook. The orderbook is generally public; users read it to place or cancel orders based on their own strategy.

Traditional asset exchanges are usually non-custodial and thus forgo the settlement layer for a settlement period, defined by the SEC, that dictates when participants of a trade must transfer their assets to the other party. This is handled by brokers who act as custodians. In contrast, the majority of cryptocurrency exchange volume happens on custodial cryptocurrency exchanges; exchanges hold users' funds and settle trades on their behalf [12].

Front-running resistant decentralized exchanges Khalil et al. [10] describe a protocol based on timelock puzzles and scoring of exchange behavior to provide resilience to front-running. This work relies on user-defined metrics to determine whether or not the exchange is front-running their orders. Specifically, the user must submit empty orders (orders that will never match but are semantically valid) to calculate a "score" to determine whether or not the exchange is front-running their orders. This requires increased bandwidth and latency on placing an order if the user would like assurance that the exchange is not front-running orders. The probability of an order not being a zero-valued order if the user sends in $k-1$ zero-valued orders and one non zero-valued order is $\frac{1}{k}$. For a user to limit the exchanges' ability to pick a non-empty to front-run with a probability of 1% she would need to send 100 orders, 99 of them empty. In other words, required bandwidth scales linearly with k.

The Injective Protocol [6] claims to provide a front-running resistant decentralized exchange service, using Verifiable Delay Functions [9] to prevent order relayers from front-running orders. Their model consists of a set of *relayers* and a set of *takers*, where *takers* are users which place orders using a smart contract that is hosted on a sidechain, or a blockchain that is allows assets from other blockchains to be swapped in and out and temporarily tracked by this sidechain by use of a smart contract. This sidechain has a set of validators, of which it's assumed that less than $\frac{1}{3}$ are byzantine. We do not require a sidechain, nor do we assume that some fraction of the users or exchange are honest.

Front-running resistant securities exchanges. Thorpe and Parkes [21] describe the design of a cryptographic security exchange that uses zero knowledge proofs to show that the exchange is matching orders correctly. The orderbook host in this work is a neutral third-party bulletin board responsible for receiving and posting orders. The exchange operator receives these orders at the same time as other users receiving updates from the orderbook host. This work only provides front-running resistance if the orderbook does not collude with the exchange operator. ClockWork does not have this requirement.

In other work Parkes et. al. [16] depends on a trusted third party *Time Lapse Cryptography* service to perform a similar function to our timelock puzzles in ClockWork. We do not require a single service, and provide a concrete instantiation for individuals to use their own timelock puzzles, which don't require coordination or a trusted third party. Rabin and Thorpe provide an instantiation of a distributed Time Lapse Cryptography service which requires a majority of the participants of the service to be honest [18]. ClockWork has no such requirement, and enjoys even simpler cryptographic assumptions.

Front-running attacks on blockchains Eskandari, Moosavi, and Clark [8] categorize front-running attacks on blockchains and discuss possible solutions and mitigations

for these attacks. This work categorizes front-running attacks into three template attacks: *displacement attacks*, *insertion attacks*, and *suppression attacks*. An adversary executing an insertion attack will observe some transaction and insert their own transaction or transactions before it. This type of attack is the most relevant to our work. The next type of attack is a displacement attack, which occurs in situations such as domain name registration, where only one transaction will successfully claim the domain name. In this case, an adversary wishes to delay any transaction which claims the domain name so they can claim it for themselves. Finally, suppression attacks are like displacement attacks but the attacker only wishes to delay incoming transactions and does not care whether or not any of their own transactions are executed - the delay is the point of the attack.

Eskandari et al. then discuss mitigations for front-running attacks. They discuss mitigations which limit the ability of some party to order transactions, giving the example of ordering transactions by their hashes' lexicographic ordering. They note that this is not sufficient for preventing front-running, but can only make it more difficult depending on the implementation. This technique would not address the problems in our setting, where the exchange could arbitrarily drop or insert orders.

Eskandari et al. review commit/reveal strategies, and also note the problem with a user potentially not revealing. They suggest a bonded commit/reveal protocol, where a user locks up funds in the commit step, and can only unlock those funds if they execute the reveal step. This is an improvement over vanilla commit/reveal, but comes with certain downsides: First, it requires locking up capital, which is costly. Second, if the adversary were posed to make more from front-running than the value of the locked funds, they would likely still take advantage of its ability to not reveal. Finally, there is a potential for miners to censor users' reveals and force them to invoke the penalty. We present a solution which does not require locking up funds, yet still allows the exchange to prove it did not front run the user.

Timelock puzzles. Many other works have used timed primitives for security [2, 3, 7]. Our protocol takes advantage of Timelock puzzles [19], which allow a user to publish data which can only be revealed or decrypted after some fixed amount of time.

Boneh and Naor introduced in Timed Commitments [4] a scheme which applies timelock puzzles to blind auctions. Their application addresses the same problem we are addressing in ClockWork: A commit-reveal protocol, where a user or auctioneer might not open their commitment. Our use of timelock puzzles is very similar to the one in Timed Commitments. We extend these ideas into a concrete protocol that combines timelock puzzles with exchange commitments and a transcript, which lets an exchange offer a record of good behavior, and users produce evidence of fraud if an exchange misbehaves. We also implement these ideas in an exchange to show they are practical.

3 Model

3.1 System Model

Participants. There are n users, $u_1, u_2, \ldots, u_n$, who want to trade digital assets. Users register with an *exchange*, $\mathcal{E}$, where they create accounts and deposit assets they intend to trade. Note that this could be a *custodial* or *non-custodial* exchange, our protocol does not distinguish between the two. For simplicity, we assume the exchange is custodial, meaning that to trade with cryptocurrencies, the user must create a blockchain transaction which sends the coins to the exchange's address and for fiat currencies, the user must make a credit card payment or bank transfer to the exchange's account. While trading, the funds are in control of the exchange. Each user has a public and private key pair (pk_i, sk_i) and the exchange has key pair (pk_E, sk_E).

There are one or many third-party *auditors* who are interested in auditing the exchange to make sure it is behaving appropriately and complying with financial regulations. A user might be an auditor.

Exchange operation. Users and exchanges have authenticated communication channels. Users create *orders*, which are offers to buy or sell a specific amount of one asset for a specific price. For example, let's say the previous clearing price for one Bitcoin was \$8. u_i might create an order to sell one Bitcoin for \$9. Users send these orders to the exchange which collects orders, aggregates them into a batch, and then runs a *matching algorithm* to match orders. Unlike the continuous order matching widely used in many crypto-currency exchanges, our exchange operates in discrete batches. The orders submitted into a batch are hidden until the batch is resolved, and orders are not given priority based on the time within the batch they are received. Once orders are matched, they are executed; for example, u_i might send its order to $\mathcal{E}$and u_j might send an order to buy one Bitcoin for \$10. We say these two orders *cross*; u_j is willing to buy for more than u_i's sell price. The exchange can match these orders and execute the trade at a certain price depending on the matching algorithm; the exchange will debit and credit the users' accounts appropriately.

3.2 Front-running

Despite the batch process treating all orders as arriving simultaneously, there are ways the exchange can exploit its information advantage over the participants to execute trades that benefit the exchange at the user's expense. In the above case, the most straightforward matching algorithm would result in a clearing price of \$9.50; both u_i and u_j would be happy with a trade, as the buyer is buying for less than the

maximum they were willing to pay, and they seller is selling for more than the minimum were willing to accept.

The exchange can perform a variety of actions to reduce this benefit to the users. One simple attack would be to delete the order from u_j, and create an order from u_E (the exchange masquerading as a user) to buy at \$9.01. The exchange knows that u_j was willing to buy at \$10, and u_e can offer to sell at \$9.99 the next auction, keeping most of the price difference in the cross for itself.

The exchange can do this because it sees orders first, before they are publicly known. It can then selectively place its own or remove orders after reading their contents.

3.3 Threat Model

Our threat model considers the users, exchange, and auditors. We assume all parties are computationally bounded and cannot forge signatures or collide hash functions. In addition, we assume all participants are limited in terms of *sequential* resources; there is a minimum amount of time ϕ to compute a step, and no participant can compute a step faster than this. All participants can accurately measure the passage of time.

Exchange. The exchange is responsible for collecting and matching users' orders; since users usually pay a fee for this service, we assume the exchange is motivated to perform this job for at least some user orders to make money. Other than that, we model the exchange as malicious. An exchange might try to manipulate user orders, delay or drop orders, or add its own orders. An exchange might also try to get information about how users are planning on trading, or generate many fake users. We assume the exchange has access to many parallel computational resources.

Users. Users are interested in trading, but are also interested in getting any possible advantage. As such, we also model users as malicious. For example, a user might submit incorrect data, try to get an advantage by seeing other users' orders before they are matched, and try to subvert matching if they find it is not in their favor. Users might try to prevent the exchange from progressing and matching orders correctly. We assume the exchange might collude with users.

Auditor. Auditors might be users participating in the protocol, and might try to frame an honest exchange to make it look as though it is acting maliciously.

3.4 Security Goals

The primary goal of the system is to implement a *front running resistant* protocol for placing orders on an exchange; intuitively this means that the exchange should not know the contents of users' orders when it makes the decision to drop users' orders or insert or change its own orders.

In order to achieve exchange accountability for front-running, we would like to achieve the following properties:

1. **Blind commitment.** The exchange commits to a batch before any participant, including the exchange, has a chance to see the contents of the orders in the batch.

2. **Binding execution.** Once an exchange commits to a batch, all valid orders in that batch *will* execute through a matching algorithm.

3. **Liveness.** Orders by non-malicious users will be executed.

Together, this implies that the exchange cannot front run, because it has no privileged information about the orders in the batch when it creates its own orders or modifies the orders in the batch.

ClockWork's second goal is to make sure that users can confirm that this is happening, and to enable the exchange to provide a third-party verifiable transcript that it did not front run. Users participating in a batch can verify and attest that their order was not front-run. An offline auditor can read the provided transcript, and determine via attestations that certain users believe the exchange did not front-run in the past.

Note that privacy is not an explicit goal: ultimately, the contents of all users' orders are revealed publicly. However, we wish to maintain privacy during a part of the protocol to prevent the exchange from front-running.

3.5 Cryptographic tools

Timelock puzzles. A *timelock puzzle* is a function which requires many sequential steps to compute. We use the repeated squaring time lock puzzle from [19]. This puzzle can be created and verified quickly if the trapdoor information is known, but takes many steps when this private information is unknown. We describe the way our protocol uses timelock puzzles in §4.1.

Digital signatures. ClockWork uses digital signatures and one-time digital signatures: $\mathsf{Sig}_{\mathcal{C}}(m) \rightarrow s$, where s is the signature on data m that is able to be verified with $\mathcal{C}$'s public key. $\mathsf{OneTimeSig}_{\mathcal{G}}(m) \rightarrow s$ is the one-time signature on data m that is able to be verified with $\mathcal{G}$'s public key.

4 Design

As described in §3.4, our goal in ClockWork is to force the exchange to commit to a batch of orders before it can see the contents; this naturally lends itself to a commit-reveal scheme. However, a simple commit-reveal scheme is not enough; this can lead to cases in which the exchange still has the ability to front run by selecting only desirable values out of a batch. It is always possible that some users might be online for the commit phase, but not the reveal. In this case, the exchange must be able to mark some of the orders as "not received" because it cannot open them. However, a malicous exchange

could simply *pretend* a user was offline and drop undesirable orders during the reveal phase. Observers would not be able to tell whether the user truly was not online for the reveal phase or if the exchange simply ignored the users' reveal.

One might consider separating the exchange into two parties: an *orderbook host* and the *matching engine*. The orderbook host is responsible only for collecting commitments and reveals and posting them publicly. The orderbook host implements a *public bulletin board* interface: it is a public, append-only log. The matching engine then looks at order commitments and reveals that are posted to the orderbook host in the relevant timeframe, and matches and executes those orders.

Unfortunately, this design does not achieve the properties we want if the orderbook host colludes with a user or with the exchange. If the orderbook host reveals the orders as the commitments are opened, the last user *still* has the choice of whether or not to reveal her own order, and can do so after seeing the orders that other users have revealed. An exchange could pose as a user and submit many different orders, revealing only the ones that are most advantageous to it.

Our key insight is that we cannot allow any party to influence order matching or execution after commitment: we force all correctly committed orders to be executed *even if* the relevant parties disappear or act maliciously later in the protocol. The rest of this section explains how we achieve this.

4.1 Timelock puzzles

In order to force open all committed orders, we revisit the idea of timed-release crypto. Timelock puzzles have an interesting property in that a party can commit to a value that is not immediately known, but will be learned some time in the future.

In ClockWork users encrypt their orders with a key that is the solution to a timelock puzzle, first introduced in [19], as well as widely used authenticated encryption from [20]. We use three functions in our protocol: Timelock, TimeUnlockFast, and TimeUnlockSlow. These functions incorporate cryptographic primitives and functions in addition to the usage of timelock puzzles.

- $m \in \{0,1\}^*$ is the message to be time-lock encrypted and is of arbitrary length and content.
- $t \in \mathbb{N}$ is the difficulty parameter which determines how long the puzzle takes to solve when using TimeUnlockSlow.
- $c \in \{0,1\}^*$ is the time-lock encrypted message and is of length proportional to m.
- $n \in \mathbb{N}$ is the RSA modulus which unlocks c in t steps.
- $p \in \mathbb{N}$ is a prime factor of n, which unlocks c in O(1) steps, regarless of t.

The functions are as follows:

- $\mathsf{Timelock}(m,t) \rightarrow (c,n,p)$. The Timelock function takes a plaintext message to be time-locked, as well as the difficulty parameter t. It first randomly generates secret prime numbers p and q, and computes the public modulus $n = p * q$. This modulus must be large enough that factorization is computationally infeasible, or approximately 2000 bits in length. The function next computes $\phi(n) = (p-1)(q-1)$, $e = 2^t \mod \phi(n)$, and $b = 2^e \mod n$. Next it takes $k = sha256(b)$ and uses key k to encrypt m with AES-GCM: $c = Enc_k(m)$. The function then returns (c,n,p).
- $\mathsf{TimeUnlockFast}(t,c,n,p) \rightarrow m$. If p is available, decryption of ciphertext c takes no longer than encryption did. Given n and p, this function computes $q = n/p$, confirms that $q \in \mathbb{N}$ and that both p and q are prime, otherwise returning $\emptyset$. It then computes $\phi(n) = (p-1)(q-1)$. With $\phi(n)$, the function can now compute $e = 2^t \mod \phi(n)$ and $b = 2^e \mod n$ just like Timelock did. With $k = sha256(b)$, this function also uses AES-GCM and computes $m = Dec_k(c)$ and returns m if the decryption is successful, $\emptyset$ otherwise.
- $\mathsf{TimeUnlockSlow}(t,c,n) \rightarrow m$. If p is not available, the decryption of m takes much longer to compute. As neither p nor $\phi(n)$ are efficiently computable from n, repeated squaring is instead used to compute $b = 2^{2^t} \mod n$. This takes t steps of modular exponentiation, reducing modulo n each time. Once b is computed, $k = sha256(b)$ allows computing $m = Dec_k(c)$ and the function returns m upon successful decryption, $\emptyset$ otherwise.

We use timelock puzzles to hide users' orders from the exchange before the exchange has committed to a set of orders to match.

While we use the repeated squaring modulo a product of primes like in the construction described by Rivest et al. [19], which we will refer to as RSW, we modify the syntax of interacting with the timelock puzzle as well as the encryption algorithm and method of creating the key. Our syntax uses a message as input to the timelock puzzle - this does not change the internal operation of the timelock puzzle, and the RSW construction could be defined this way as well. More importantly, we use AES-GCM rather than RC5 to encrypt the message and obtain the ciphertext. The original RSW construction adds the RC5 key to the value $b := 2^e \pmod n$, whereas we hash the value b and use that as the key for AES. This way, one will still obtain the key for order decryption.

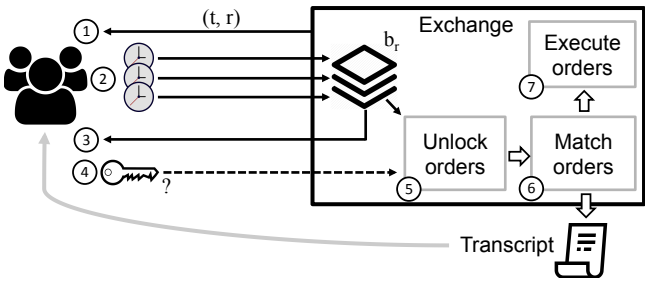

Figure 1: ClockWork architecture. An exchange begins a round (1). Users submit orders encrypted in a timelock puzzle to an exchange (2), which collects the orders into a batch and sends them out to users (3). Users may or may not send the trapdoor to unlock the puzzle quickly (4). The exchange unlocks the orders (5), and then matches (6) and executes them (7). The numbered steps correspond to the protocol described in §4.3

4.2 Sending and matching orders

Figure 1 shows the high-level architecture and protocol of ClockWork. In §4.3 we give a concise summary of the protocol. Users $u_1, \ldots, u_n$ create and send orders to the exchange $\mathcal{E}$. Orders in ClockWork are executed in rounds of batches in order to create a commit point for a set of orders. During a round, the exchange publishes a new *batch id*, and the *batch window* begins. During this time, users submit orders for the batch. At the end of the window, the exchange commits to orders for a batch and gathers evidence that it did not front run for its transcript. Then, the exchange opens the batch, after which it can match and execute all correct orders. Note that opening the batch does not require cooperation from the users, but it is more efficient if users cooperate.

Setup. Users register with exchange and place funds in deposit to trade over multiple rounds. Everyone agrees on a public difficulty parameter, t, where $\Delta = t\phi$ is the minimum amount of time we wish the puzzle to take to compute. We discuss how to appropriately set t in §4.4. The exchange should also publish its deterministic matching algorithm $\mathcal{M}$, and its deterministic rules for determining if an order is well-formed.

Begin round. The exchange generates a unique round number r for this batch, which it then signs along with t and broadcasts to all users.

Send orders. Users create orders for this batch; for simplicity of exposition we assume each user creates one order, though in practice each user might create multiple orders. Exchanges might have different types of orders. Our only requirement is that user i's order, o_i, contains enough information for the exchange to execute the order completely (it should not require more interaction with the user) and that the order includes r. The user encrypts the order inside a timelock puzzle which, as described in §3.5, yields a ciphertext of the encrypted order, p_i that can be used to compute the answer to the puzzle quickly, and the modular n. The user sends the encrypted order and n (which we call puzzle_i) and a signature on this data to the exchange, but keeps p_i stored locally. The user also stores the timestamp of when it sends the order.

Commit. The exchange verifies the signature on each puzzle it receives. It selects some subset b_r of these puzzles to be part of the batch. The exchange sends b_r and $\mathsf{OneTimeSig}_E(b_r)$ to the users selected for the batch. This serves to select the set of orders that *will* be executed for this batch r. Via this signature, the exchange is committing to the orders, timelock puzzles, and user signatures on the timelock puzzles. Note that if the exchange wishes to know if it could advantageously insert new orders or include or drop an order based on its contents, it would need to have solved orders' timelock puzzles by now.

Attest to exchange response time. Users who have their orders selected for the batch sign b_r and send their signatures $\mathsf{Sig}_i(b_r)$ back to the exchange *only* if they received the b_r within Δ time of sending puzzle_i, which is the minimum amount of time the user believes it will take to compute the puzzle. Note that users should *not* sign if they believe the exchange has had enough time to solve the timelock puzzle and decrypt the order before sending out its commitment to b_r and signature. Because of the timelock puzzle, the user knows that the exchange did not see the contents of her order before committing to the batch, and thus the exchange could not have made a decision to insert, drop, or edit any orders based on the information in her order.

A misbehaving user might not send a valid signature, and we explain how we tolerate that below. At this step, users should *also* send the trapdoor to the puzzle, p_is. However, we will explain what happens if a user does not.

Open. The exchange needs to make sure the timelock puzzle is wellformed; for each submitted p_i it should test that p_i and n_i/p_i are prime. If all users submitted the correct p_i's, then at this point the exchange can solve the puzzles quickly using the trapdoor, decrypt all of the orders and proceed with matching and execution. We call this the *fast path*. If some user i did not respond with a p_i or using the p_i does not decrypt the order, then the exchange must solve the timelock puzzle in puzzle_i in order to decrypt i's order. We call this the *slow path*. If, after computing the timelock puzzle, the order does not decrypt, then the exchange should not include this order in the batch. Otherwise, even if p_i does not work, if it can solve the timelock puzzle on the slow path it should include the order. The intuition behind this is that the time at

which the user reveals p_i is *after* the exchange has committed to the batch. The user should have no way of changing whether or not an order is accepted after that point.

Assuming all the orders were well-formed (we will discuss what to do when an order is not well-formed or does not decrypt), the exchange now has the following:

- A batch b_r of puzzle_i's, each of which includes a valid signature on puzzle_i by user i.
- A set of decrypted user orders for round r, one for each puzzle_i in b_r.
- A set of user signatures on b_r. We might not have every user's signature.

The exchange runs matching algorithm $\mathcal{M}$ on all *valid* decrypted orders, and executes the matched orders.

Order validity. An order contains a batch number, r, and is only valid for that batch. If the decrypted order contains a different r, the exchange should not match the order. If the timelock puzzle does not correctly decrypt an order, there is no way for any party to decrypt the accompanied encrypted order, and it cannot be matched. Similarly, if the information in the decrypted order is purposefully useless data, that cannot be matched as an order either. The user may have also submitted an order which is in the correct format, but they do not have the funds to fulfill. This would also be invalid and can't be matched. There are many more cases like this that depend on the specific order matching rules, and ultimately the valid orders would be matched and settled by the exchange. All that we require is that these rules are deterministic and are published for anyone to verify.

In summary, an order is considered invalid if any of the following occur:

- The order's timelock puzzle does not decrypt to a valid order in the slow path
- The order o_i is not well-formed
- The user signature on the order's puzzle_i is not valid
- The user does not have the funds at the exchange to execute the order

A user does not have to reveal p_i or sign b_r for the order to be valid. In fact, even if the user sends a bad p_i, if the exchange can correctly solve the puzzle_i in the slow path, it should still include o_i, but not include the bad p_i in the transcript.

4.3 Protocol

In summary, the protocol is as follows:

1. $\mathcal{E}$ begins a round by broadcasting (t,r)

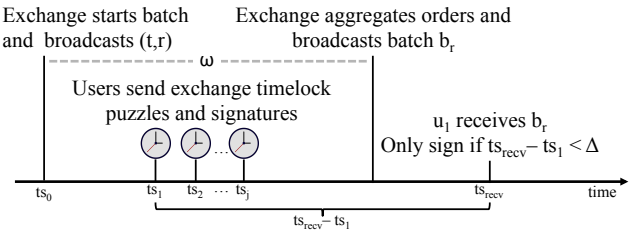

Figure 2: The timeline for order submission and commitment, and a safety condition for when a user should sign b_r. The user should only sign if they are confident that less than Δ time has passed since the exchange has received their the order.

2. u_i creates order o_i and generates $\mathsf{Timelock}(o_i,t) \to (c_i,p_i,n_i)$. The user saves p_i and the current time as ts_i and sends puzzle (c_i,n_i) and $\mathsf{Sig}_i(c_i,n_i)$ to the exchange.
3. $\mathcal{E}$ receives (c,n,s) and checks the signature. $\mathcal{E}$ creates a list of all received puzzles with valid signatures, b_r. It broadcasts b_r and $\mathsf{OneTimeSig}_{\mathcal{E}}(b_r)$ to all users with orders in the batch.
4. u_i receives (b_r,s). She checks that her order is in b_r, that the signature s is valid, and that $now - ts_i < \Delta$. If so, u_i sends $\mathcal{E}$ $(p_i, \mathsf{Sig}_i(b_r))$
5. If $\mathcal{E}$ receives p from a user with an order in the batch, it computes $\mathsf{TimeUnlockFast}(c,n,t,p) \to o$. Otherwise, or if that fails, it computes $\mathsf{TimeUnlockSlow}(c,n,t) \to o$. If this fails, $\mathcal{E}$ discards the order.
6. $\mathcal{E}$ matches the remaining valid orders and creates a $\mathsf{transcript}_r$ for the batch.
7. $\mathcal{E}$ executes the matched orders.

4.4 Setting timelock puzzle difficulty

The exchange must set t appropriately. If t is too small, then message delay could prevent users from signing b_r, as Δ will have passed too quickly. If t is too large, then slow path timelock puzzles will take a long time to open, delaying batch matching and settlement. Note that the exchange does not have to wait for one batch to settle before starting another one, but users might prefer to know the outcomes of previous batches before submitting new orders.

Setting t depends on maximum expected message delay between participants, δ, and the batch window length, ω. The size of the batch window is dependent on the market and participants; for example low-latency capital markets might have ω on the order of seconds or hundreds of milliseconds [5]. The longer the window, the higher t must be in order to tolerate different puzzle arrival times. Figure 2 shows the safety condition for the user that sends its order first; $t_{recv} - ts_1 < \Delta$.

The exchange does not know exactly when this order was sent (it does not know ts_1) but it does know ts_1 must be after when it broadcast (t,r) (ts_0) so the exchange must end the batch and broadcast b_r well before $ts_0 + \Delta - \delta$ so that b_r is guaranteed to arrive before the first user experiences Δ time passing. This means $\omega < \Delta - \delta$ or t should be much greater than $\frac{\omega+\delta}{\phi}$.

4.5 Tolerating misbehaving users

ClockWork tolerates misbehaving users. There are many different ways that a user might misbehave, and here we describe how ClockWork's design ensures that these misbehaviors will either be prevented or detected.

First, a user might submit a malformed order. The exchange should not match this order but should include the order and the user's signature on the order in the transcript so any auditor can verify the order was malformed. Order validation rules should be deterministic.

A user might not reveal p_i after an exchange has committed to execute the order by including it in b_r. In this case, the exchange will have to take the slow path and solve the timelock puzzle in order to decrypt the order. If everything else about the order is well-formed, then the exchange must match this order along with all the other orders in the batch.

A user might submit the incorrect p_i. In this case, the exchange must fall back to the slow path, and if the slow path works, and the order is well-formed, then the exchange *must* execute the order, even if it does not decrypt correctly on the fast path.

If the exchange cannot decrypt the order on the fast or slow path, then it should not match the order, *even if it is able to eventually obtain access to the plaintext order* (e.g., the user eventually reveals it to the exchange.

A user might do everything correctly but not sign the batch commitment b_r. In this case, the exchange must still include the user's order since the exchange committed to it in b_r. Unfortunately, the exchange will not enjoy the user's affirmation that the exchange did not front run, and if all users are malicious, an offline auditor will not be able to determine whether or not the exchange did front run orders in b_r.

Finally, a user might submit a correct order, but the user might not have the funds on deposit at the exchange to actually execute the order. In this case, the exchange should mark the order as such in the transcript and refuse to execute the order. In a custodial exchange, the exchange would have to open its books to the auditor to prove that this was in fact the case. In a non-custodial exchange, the exchange could point to insufficiently funded channels or smart contracts on the relevant blockchains.

Algorithm 1 Procedure to validate the transcript for batch r.

```
1:  procedure VERIFY(transcript_r, s)
2:      b_r ← transcript_r.batch
3:      t ← transcript_r.t
4:      r ← transcript_r.r
5:      if ¬ VERIFYSIG(E, transcript_r, s) then return false
6:      for all (c_i, n_i, s_i, p_i, o_i, exec_i) ∈ b_r do
7:          if p_i then
8:              o' ← TIMEUNLOCKFAST(c_i, n_i, t_i, p_i)
9:              if o' = ∅ then return false
10:         else
11:             o' ← TIMEUNLOCKSLOW(c_i, n_i, t_i)
12:         if exec_i then
13:             if o' = ∅ then return false
14:             if ¬VERIFYSIG(i,(c_i,n_i),s_i) then
15:                 return false
16:             if o' ≠ o_i then return false
17:             if ¬VALIDATEORDER(o_i, r) then
18:                 return false
19:         else
20:             if VERIFYSIG (i,(c_i,n_i),s_i) then
21:                 if o' = o_i then
22:                     if VALIDATEORDER(o_i, r) then
23:                         return false
24:     return true
```

4.6 Transcripts and offline verification

The exchange creates a $\mathsf{transcript}_r$ for round r. This serves as a record of execution. The exchange makes this and $\mathsf{OneTimeSig}_E(\mathsf{transcript}_r)$ publicly available (for example, on its website). Once published, anyone can audit the transcript to verify that orders were processed correctly and matched.

The transcript consists of the following pieces of information:

- t, r, b_r, $\mathsf{OneTimeSig}_{\mathcal{E}}(b_r)$
- For each order in b_r:
  - $c_i, n_i, s_i = \mathsf{Sig}_i(c_i, n_i)$
  - o_i if the puzzle decrypted correctly, otherwise nil
  - p_i if the user sent a correct p_i, otherwise nil
  - *exec*, whether or not the exchange plans to execute the order
  - $\mathsf{Sig}_i(b_r)$ if the user sent a signature, otherwise nil

Both users who were online during batch execution and offline auditors might refer to the transcript. Users will confirm that the exchange is indeed committing to the batch that the user saw for round r. Any user can check the signatures and run the puzzles themselves (if necessary) to determine the set of decrypted orders, and can check each order for validity to

get the set of orders that should be matched. The user can then run $\mathcal{M}$ on the set of valid orders and make sure they get the same result for their own order as what the exchange actually executed and settled. If at any point validation of the transcript fails, the user can publish evidence of this failure and alert the appropriate authorities. Algorithm 1 shows the procedure used to validate the transcript.

Note that the transcript verifier needs to check both executed *and* unexecuted orders. Otherwise, the exchange could claim orders in b_r that should have executed were invalid; the verifier must see that this is true.

In the protocol described so far, a puzzle with a bad trapdoor p but which decrypts successfully in the slow path must be executed. This is because p is only revealed later, after the exchange or a curious user might have had a chance to see other puzzles. It is too late to insert new orders at this point (the exchange has already signed the batch) but if we were to have an incorrect p cause the order not to execute, then the exchange (or another user) could cancel disadvantageous orders by supplying a bad p. This breaks our binding execution goal in §3.4. No party should be able to affect whether or not an order will execute *after* the batch commitment point.

This transcript will include puzzle trapdoors for all orders where users supplied them; in this case, well-formed order execution is quickly verifiable by a third party. However, if the user did not supply a trapdoor, the exchange cannot provide a quickly verifiable proof. The verifier will have to try to solve the puzzles themselves and confirm whether or not the puzzle decrypts to a valid order.

When the exchange claims the timelock puzzle was invalid, the verifier must also try to solve the puzzle to confirm it is invalid. If it turns out a timelock puzzle the exchange claimed was bad *was* well-formed, the exchange has failed verification. The verifier can raise a concern, but any other verifier would also have to solve this puzzle to confirm. The exchange's signature on the transcript is evidence of an incorrectly dropped order. In §4.7 we describe how a different type of timelock puzzle might let us provide faster proofs of incorrectness.

Multiple commitment consequences. A malicious exchange might try to partition the set of users to create multiple batches for the same round r, and choose between the batches to find the most advantageous. This is another form of front running. However, this requires the exchange to sign two different batches, b_r and b'_r. Because we require one time signatures, this would reveal the exchange's private key, which is evidence of misbehavior.

Assurances. The Clockwork protocol provides assurances for different entities. For users participating in an auction, it provides assurance that their order was included in the auction without knowledge of the order contents. It does *not* provide the same assurance with regards to other users' orders, even if they provided $\mathsf{Sig}_i(b_r)$; they may have colluded with the exchange, or they may themselves be the exchange.

This issue of Sybil-users is even more relevant when the entity that wants assurances is a third party observer that did not participate in the batch. Observers may want to inspect the transcripts of previous auctions to see if an exchange is legitimate in order to decide if they will join and trade on an exchange. However, a malicious exchange could simulate an auction, by playing the part of many users and adhering to the protocol, so that the transcript checks out. In fact, such a simulation has much lower computational requirements when compared to a real auction, as the exchange has created every puzzle and thus knows the private modulus factors. This means no iterative computation is required, allowing a malicious exchange to quickly create years worth of historical auctions.

In order for an observer to be convinced that the transcripts are valid, they need to know that the users are distinct from the exchange and trustworthy. In the most direct case, the observer's friends use the service, and they don't think their friends are colluding with the exchange. Exchanges still can populate an auction with many Sybil users to give the appearance of a larger user base, but observers should not be convinced by large numbers of traders.

On an exchange regulated by a trade association or government, auditors may be able to inspect customer identities to resolve the issue of exchange-created sybils. If an auditor is convinced that all the participants of an auction are legitimate, the auction transcript will give evidence that no front-running occurred during the batch.

4.7 Optimizations

Merkle tree commitments. Instead of sending the entire batch to each user in step 4 of the protocol (§4.3), the exchange can create a merkle tree from the orders in b_r and send each user the root of the tree and an inclusion proof for their order. This is enough to convince each user that their order has been included in b_r, and later on the users can confirm that the same b_r is included in the transcript.

Including the trapdoor in the order. Another optimization is to include the trapdoor, p, in the encrypted order. This changes the signature for the timelock puzzles and decrypt functions. The exchange would consider orders without the correct p invalid. The verifier still needs to perform $TimeUnlockSlow$ on invalid orders to confirm that p is missing, but it acts as a disincentive for users to omit p as their orders would be invalid; with this optimization verifiers only need to run $TimeUnlockSlow$ on orders submitted by malicious users, not honest users with poor network connections.

Verifiable Delay Function proofs. ClockWork uses RSW [19] timelock puzzles. As described so far, the user reveals

the factoring of the modulus, p_i, used to constrcut the time-lock puzzle, and the exchange publishes this in the transcript. This means that for an order in the transcript which the exchange claims they never received p_i, the auditor has to run the slow path to confirm if the order should be included or not. ClockWork could instead require the exchange to include Wesolowski [22] or Pietrzak [17] VDF proofs for the computation done by the exchange to solve puzzles and decrypt orders. For example, the exchange could include the value $b = 2^{2^t} \pmod n$ and a proof π_i that b actually does equal $2^{2^t} \pmod n$. An honest exchange could do this so users may verify that it has not lied about the timelock puzzle solution, and the users do not need to compute b in t time on their own. If an order is not properly formed (e.g. $\mathsf{Decrypt}_k(c)$ is random data) This would benefit users in verifying that an order is not properly formed, as the exchange can give users a b and a proof π_i, users can verify the proof and finally see that the decryption does in fact return a non well formed order.

5 Security Analysis

To analyze the security of our protocol against front-running, we first must define the types of attacks and behavior that a malicious exchange or other adversary could carry out which we would call "front-running".

5.1 A security game for online adversaries

We first define a security game for online adversaries. These adversaries are "online" because they play the part of the exchange in the protocol and must interact with a batch participant, which we will call the challenger. This adversary's goal is to front-run the user, and we define this as being able to tell whether or not this order is a "real" order (not random data), and the adversary will output 1 if it thinks it is a real order, and 0 otherwise. The adversary also would like to convince the user that it has not been front-run, so we say that the adversary automatically loses the game if it responds in a time greater than Δ.

We say that the adversary succeeds if the FR_b game outputs 1, with $b = 1$ representing the "real" game and $b = 0$ representing the "random" game. The challenger will pick an order from O, which is the set of well-formed orders, and we assume these orders have length l. We define a function $\mathsf{CurrentTime}$ which is an oracle that returns the current wall clock time. The batch parameters r,t will be globally available to the adversary and challenger. We give the challenger $\Delta = ct$.

Letting W be the event that the adversary "wins" the game (the output of $\mathrm{FR}_b = b$), we say that the protocol is secure against front-running if:

$$\mathrm{Adv}^{FR}(\mathcal{A}) = \left|\Pr[W] - \frac{1}{2}\right| \leq \varepsilon$$

Algorithm 2 FR_b

procedure FR_b
  if $b = 1$ **then**
    $\mathbf{o_i} \leftarrow O$
  else
    $\mathbf{o_i} \xleftarrow{\$} \{0,1\}^l$
  $(c_i, n_i, p_i) \leftarrow \mathsf{Timelock}(\mathbf{o}, t)$
  $s \leftarrow \mathsf{Sig}_i(c_i, n_i)$
  $starttime \leftarrow \mathsf{CurrentTime}()$
  Send (c_i, n_i, s_i) to $\mathcal{A}$.
  On receiving b' from $\mathcal{A}$:
  $endtime \leftarrow \mathsf{CurrentTime}()$
  if $\Delta < (endtime - starttime)$ **then**
    return $\neg b$
  return b'

5.2 ClockWork's security

In this section we consider ClockWork's security against an adversary that wishes to front-run a batch participant. We will now assume that there is an adversary which can win the game with a non-negligible probability, or that there is an adversary which can determine whether or not the order $\mathbf{o}_i$ is a random string in less than Δ time.

This would mean that, using only n, c the adversary can determine whether or not the underlying order is real or random. Given that n is an RSA modulus and c is the output of a symmetric encryption algorithm, then either:

1. The adversary can distinguish plaintext o from a random string using only ciphertext c, bypassing the time-lock

2. the adversary can compute some function on c, n in time less than Δ which it can use to determine if the order is real or random.

In the first case, this would imply that the adversary could break the encryption algorithm used for $\mathsf{Encrypt}$, so because we assume the encryption algorithm is semantically secure, this is not possible.

In the second case, the adversary would like to compute the timelock puzzle in order to distinguish the order from random. There are two known methods computing k: computing 2^{2^t} mod n by repeated squaring or factoring n in order to use the $\phi(n)$ shortcut. Assuming the symmetric encryption is secure, the only ways the adversary can distinguish the order from random is either solving timelock puzzle in less than Δ time, or factoring the modulus in less than Δ time. We assume n to be a safe RSA modulus, and we assume that the adversary cannot solve the timelock puzzle in less that Δ time so these attacks are not possible either.

6 Implementation

We created a framework to implement experimental features for cryptocurrency exchanges, and have since implemented the front-running resistant timelock puzzle scheme, where the exchange is the orderbook host. Our (anonymized) code is available at https://gitlab.com/clockwork-paper/clockwork. The exchange responds to queries for the current auction with a batch ID, and publicly broadcasts auctions it has committed to matching. It then matches batch auctions when the timelocked orders have been solved, and publishes the results. We wrote the library in Go, and implemented the RSW timelock puzzle scheme in Go as well. The implementation uses ECDSA signatures and the Noise protocol framework to provide authenticated and encrypted communication.

7 Evaluation

In our evaluation we focus on answering a few questions to do with the scalability and cost of deploying our protocol:

1. In the best case, where all users respond with the timelock puzzle solution, does the protocol incur reasonable overhead?
2. In any other case, where the exchange must solve a timelock puzzle, what is the overhead for the protocol?
3. If solving many timelock puzzles benefits from utilizing multiple cores, does it also benefit from GPU use?

7.1 Setup

There were two main system setups which we evaluated our system on. In order to evaluate the difference in throughput between a GPU and CPU, we used a Google Cloud Platform N1-Standard-1 instance equipped with a Tesla V100, a single-core CPU, and 3.75GB of RAM. All arithmetic operations on the CPU used the GMP library, whereas the GPU was used to run many parallel calls to the modular power operation in the CGBN[1] library.

To evaluate solution rate with unanimous responses, we used a machine with a quad-core Intel i7 6700HQ, and 16GB of RAM. This machine also used the GMP library and the Golang standard big number package for arithmetic operations.

7.2 Transcript size

The transcript size depends on the number of participants in the order batch. Verification requires checking a fixed number of signatures for each user, and confirming that the results published by the exchange are actually the result of solving the timelock puzzles submitted by users. The exact size of the transcript depends on the parameters chosen for timelock puzzles and orders. In our implementation, the moduli needed for timelock puzzles are notably large components of the transcript. We evaluate the protocol using a 4096-bit modulus. Given the vast majority of implementations of this protocol would likely set puzzle parameters so the puzzle is solvable on the order of minutes or seconds, one could easily use a 2048 or 1024-bit modulus in order to reduce transcript size. This is reasonable as long as the exchange or another adversary would have a negligible probability of obtaining the modulus' factors in the time it takes to solve the timelock puzzle.

7.3 Evaluation of protocol overhead

In order for the system's performance to be comprehensively evaluated, there are two scenarios that we are concerned with: Either every user in the batch responds with a signature and correct trapdoor, or at least one user is not responsive or responds with an incorrect trapdoor. If every user responds, then the exchange does not need to solve any timelock puzzles by repeated squaring. Given the trapdoor p_i, the exchange only needs to compute $2^t \pmod{\phi(n_i)}$, and use that value to decrypt the order ciphertext.

Unanimous Response Performance. Compared to the case where not all users submit a signed exchange commitment and trapdoor, the exchange needs to perform exponentially less squarings in order to obtain the key for each user's order ciphertext. If this is the case, then our system should show that increasing the timelock puzzle parameter should not decrease the puzzle throughput. As shown in Figure 3, this is certainly the case, with puzzle throughput remaining roughly the same for large batch sizes while increasing the timelock puzzle difficulty parameter by multiple orders of magnitude.

Non-unanimous Response Performance. When not all users submit a signed exchange commitment and correct trapdoor, the exchange must solve the timelock puzzles which don't have trapdoor responses. This means that the latency for solving a batch should increase linearly with the timelock puzzle difficulty parameter. The latency should also increase if there are not enough cores to solve all orders without responses at once. To evaluate the latency in this case, we vary the timelock puzzle parameter, number of cores the exchange has access to, and the number of timelock puzzles in the batch.

We expect the latency to be dependent only on the timelock puzzle difficulty parameter if the exchange has one or more core for each order in the batch. It should not take longer to solve the batch if the exchange has more than one core per order, than if the exchange has exactly one core per order. If the exchange does have at least one core per order, then the

[1] This was created by Nvidia Research, and can be found at https://github.com/nvlabs/cgbn.

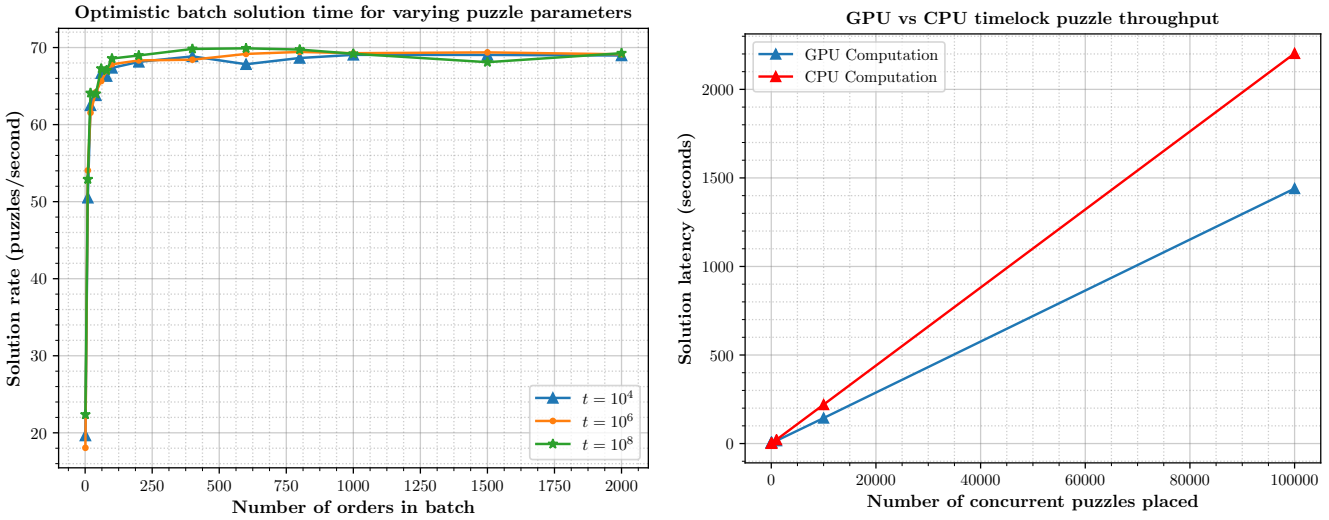

Figure 3: The measured solution rate (left) given that all users receive the commitment and respond with the puzzle answer, as per the design of the fast path. We increase the puzzle parameters by two and four orders of magnitude to show that the solution time does not vary significantly if users and the exchange follow the protocol. Comparison of CPU and GPU throughput (right) for a varying number of concurrent puzzles being solved.

latency should be dependent on the timelock puzzle difficulty parameter, and the ratio between the number of orders in the batch and number of cores available.

7.4 GPU Puzzle Solving Performance

One possible critique of the ClockWork protocol is that since each order is a timelock puzzle, and the fastest way to solve a timelock puzzle is fully occupy a CPU with solving the puzzle, the throughput of the exchange would be bottlenecked by the number of CPUs it has access to.

Motivation for GPU use. While the fastest way to solve a single timelock puzzle would be to use a CPU with high clock speed, it's reasonable to think that a GPU could instead be used to obtain high throughput of solving many simultaneous puzzles. It would seem that a GPU is well suited to high throughput timelock puzzle evaluation as the puzzles have the same time parameter, and the same operations are being run on each puzzle. Furthermore, latency increases are not very problematic for this system: they do not decrease the security of the system, but instead just increase the delay between when an order is placed and when the results of the auction are published.

GPUs fall short. We evaluate the difference in throughput for a GPU and CPU in Figure 3. This test computed the RSW puzzles $b = 2^{2^t} \pmod N$, where $t = 16384$, and N is a 4096 bit RSA modulus. While the GPU does perform slightly better than the CPU, it may not actually yield higher throughput unless significant optimizations are made to the arithmetic library and timelock puzzle solution code used. The Nvidia Tesla V100 was able to beat the single-core CPU on the GCP instance in throughput, however it did not have a significant advantage: when given 100,0000 puzzles, the CPU was solving puzzles entirely sequentially due to its single core, and the GPU had less than twice the throughput of the CPU. This could be for a variety of reasons - GMP is an exceptionally fast library for performing the arithmetic operations needed for timelock puzzle evaluation, and the library used for arithmetic on the GPU would need much more optimization in order to keep up.

8 Future work

In ClockWork, the exchange has to, in the worst case, perform an evaluation of a timelock puzzle for every order in a batch. In our protocol, we require $O(n)$ processors for n orders to address this worst case; the exchange must solve the puzzles in parallel. Though we investigated the use of GPUs and found them wanting, there might be other customized hardware that makes this cheaper to execute in parallel.

In order to verify a batch was processed correctly, any auditor must look at every order in the batch. It might be possible for the exchange to provide a more compact proof of correct execution instead of requiring auditors to perform the computation themselves.

Finally, ClockWork does not provide privacy; eventually all orders become public. It might be desirable to have unmatched orders stay private. Future work could investigate using cryptographic primitives that do not unlock unless the order is matched.

9 Conclusion

We introduce ClockWork, an exchange protocol for producing proofs of non front-running. ClockWork provides a method for the exchange to prove to a user that it did not front-run their order, while creating a transcript an exchange can use to show an auditor to claim it did not front-run. We hope this encourages more work in auditable and verifiable financial infrastructure.

References

[1] Anyone else getting constantly front-runned? Reddit.

[2] Mihir Bellare and Shafi Goldwasser. Encapsulated key escrow, 1996.

[3] Mihir Bellare and Shafi Goldwasser. Verifiable partial key escrow. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 78–91, 1997.

[4] Dan Boneh and Moni Naor. Timed commitments. In *CRYPTO*, 2000.

[5] Eric Budish, Peter Cramton, and John Shim. Implementation details for frequent batch auctions: Slowing down markets to the blink of an eye. *American Economic Review*, 104(5):418–24, 2014.

[6] Eric Chen and Albert Chon. Injective protocol: a collision resistant decentralized exchange protocol, 2018.

[7] Cynthia Dwork and Moni Naor. An efficient existentially unforgeable signature scheme and its applications. In *Annual International Cryptology Conference*, pages 234–246. Springer, 1994.

[8] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: front-running attacks on blockchain, 2019.

[9] Dan Boneh, Joseph Bonneau, Benedikt Bünz, Ben Fisch. Verifiable Delay Functions. Cryptology ePrint Archive, Report 2018/601, 2018. https://eprint.iacr.org/2018/601.

[10] Rami Khalil, Arthur Gervais, and Guillaume Felley. Tex - a securely scalable trustless exchange. Cryptology ePrint Archive, Report 2019/265, 2019. https://eprint.iacr.org/2019/265.

[11] Valentina Kirilova. Crypto trading 'bots' caught arbitraging investors by front running. LeapRate.com, 2019.

[12] Bitwise Asset Management. Bitwise asset management presentation to sec, 2019. https://www.sec.gov/comments/sr-nysearca-2019-01/srnysearca201901-5164833-183434.pdf.

[13] New York State Department of Financial Services. DFS fines Credit Suisse AG $135 million for unlawful, unsafe and unsound conduct in its foreign exchange trading business. Press Release, Consent Order, November 2017. https://www.dfs.ny.gov/about/press/pr1711131.htm.

[14] The United States Department of Justice. Former global head of HSBC's foreign exchange cash-trading found guilty of orchestrating multimillion-dollar front-running scheme. Press Release, October 2017. https://www.justice.gov/opa/pr/former-global-head-hsbcs-foreign-exchange-cash-trading-found-guilty-orchestrating.

[15] The United States Department of Justice. HSBC Holdings plc agrees to pay more than $100 million to resolve fraud charges. Press Release, January 2018. https://www.justice.gov/opa/pr/hsbc-holdings-plc-agrees-pay-more-100-million-resolve-fraud-charges.

[16] David C Parkes, Michael O Rabin, Stuart M Shieber, and Christopher Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy auctions. *Electronic Commerce Research and Applications*, 7(3):294–312, 2008.

[17] Krzysztof Pietrzak. Simple Verifiable Delay Functions. Cryptology ePrint Archive, Report 2018/627, 2018. https://eprint.iacr.org/2018/627.

[18] Michael O Rabin and Christopher Thorpe. Time-lapse cryptography. 2006.

[19] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report Technical memo MIT/LCS/TR-684, MIT Laboratory for Computer Science, 1996. (Revision 3/10/96).

[20] Joseph A. Salowey, David McGrew, and Abhijit Choudhury. Aes galois counter mode (gcm) cipher suites for tls. Technical report, 2008.

[21] Christopher Thorpe and David C. Parkes. Cryptographic securities exchanges. In Sven Dietrich and Rachna Dhamija, editors, *Financial Cryptography and Data Security*, pages 163–178, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[22] Benjamin Wesolowski. Efficient verifiable delay functions. Cryptology ePrint Archive, Report 2018/623, 2018. https://eprint.iacr.org/2018/623.