Injective Protocol: A Collision Resistant Decentralized Exchange Protocol

Eric Chen[1,3] and Albert Chon[2,3]

[1]New York University
[2]Stanford University
[3]Injective Labs, {eric,albert}@injectiveprotocol.com

December 2018

Abstract

We introduce Injective Protocol, a collision and front-running resistant decentralized exchange protocol on the Ethereum network that integrates *verifiable delay functions* (VDF) as a *proof-of-elapsed-time* to resolve same-block order conflicts while preventing front-running attacks. Our proposal is the only decentralized exchange protocol that is fully trustless, publicly verifiable, resolvable, liquidity neutral, and front-running resistant.

Contents

1 Introduction

Since the early days of cryptocurrency, major centralized cryptocurrency exchanges such as Mt. Gox [1] have been hacked, resulting in the loss of hundreds of millions of dollars worth of cryptocurrency. Such occurrences present a clear need for decentralized exchanges where users act as their own custodians for their funds. Today, modern decentralized exchanges on the Ethereum network are prone to numerous security vulnerabilities and have several design flaws which hamper full decentralization and negatively impact user experience. In this paper, we introduce Injective Protocol - a collision and front-running resistant decentralized exchange settlement logic protocol on the Ethereum network. We utilize *verifiable delay functions* (VDF) [2] as a *proof-of-elapsed-time* and *fixed-delay time-lock puzzle* to resolve same-block order conflicts and to prevent front-running attacks. Our protocol is comprised of two components: a settlement logic layer and a trustless relayer network protocol which allows for sharing of liquidity amongst distinct relayer liquidity pools and integrates with 0x[3]. More specifically, Injective Protocol is protocol-agnostic and satisfies the following properties:

- *Trustless*: The settlement logic does not require a trusted third party or a centralized trade execution coordinator (TEC) to fairly establish the true sequence of incoming orders.
- *Resolvable*: Conflicting orders submitted in the same block can be fairly and deterministically resolved through the settlement logic
- *Publicly Verifiable*: Incoming orders submit time proofs which the public can use to efficiently verify that a fair order sequence was executed.
- *Liquidity Neutral*: The protocol does not make any restrictions on accessibility of different liquidity pools and allows for open exchange.

We also propose a front-running proof model that offers these additional properties:

- *Front-Running Proof*: A predatory front-runner cannot intercept incoming orders and manipulate the sequence of order-filling to gain price advantage.

2 Background

2.1 Current State of Decentralized Exchanges

The stateful nature of the Ethereum blockchain results in a time delay between every state (i.e. each block being mined), which allows for race conditions to manifest between transactions. In decentralized exchanges with open orderbooks, trade collisions occur when transactions within the same block contain conflicting orders. As a result, decentralized exchanges such as EtherDelta only have around a 79%-90% success-rate for trade executions [4] as a direct result of trade-collisions. Such issues have led decentralized exchanges to adopt different measures to prevent collisions but all current implementations make numerous tradeoffs in *decentralization*, *trustlessness* and *liquidity*. Left unresolved, these issues weaken the value proposition of using a decentralized exchange

2.2 Decentralized Exchange Terminology

Within an exchange, there are two parties: makers and takers. Makers submit make orders which specify an offer to a specified amount of one token for another amount of another token. Takers view the exchange orderbook (which is solely comprised of make orders) and exchange tokens by filling make orders.

A ***trade collision*** occurs when multiple takers attempt to take the same make order at the same time. Since only one take order can fill the make order (assuming a simplified model of no partial fills), there is a failure in trade facilitation which results in one taker having his order fail. In an ideal DEX, trades (a take order filling a make order) would clear immediately and the orderbook would be updated accordingly, thus signaling to other traders that the make order in question is unavailable. However this is impossible for an on-chain orderbook on the Ethereum network, as orders are submitted as transactions which are included together in a block which then updates the orderbook once the block is mined into the blockchain. Since blocks are mined around every 15 seconds, DEX protocols such as 0x are prone to collisions for orders submitted within the same block period. 0x relayers have attempted to alleviate this issue by employing a continuous, centralized server that updates the orderbook displayed to the user every time a trade is submitted through their interface. However, since trade settlement is still enforced through smart contracts on Ethereum, this mechanism only prevents collisions from traders who use that relayer. Traders from other relayers targeting the same make order will still face a collision because the order status is not transmitted to them. Worse yet, front-runners can front-run the orders pending settlement regardless of the relayer orderbook status.

Front-runners leverage trade collisions to profit from intercepting take orders in DEXs. Since miners of Ethereum process transactions on a gas-time priority basis, race conditions occur when the Ethereum mempool exceeds the network's maximum throughput. Because miners prioritize transactions with the highest gas fees, a front-runner can exploit this feature by observing a large pending trade in the mempool and then submitting a colliding trade with a higher gas fee. Since the miner will likely include the front-runner's trade before the large trade, the front-runner can profit from the rise in price as a result of the execution of the large trade. The taker that was front-run would then experience order failure and would have to resubmit another take order which will likely be at a worse exchange rate (and would again be susceptible to front-running).

At scale, it is clear that the frequency of collisions would increase as trading volume on DEXs increase. In order for DEXs to support both a high frequency of orders and high volumes within orders, collisions must be minimized and the threat of front-running must be eliminated. Otherwise, high volume traders face the risk of not only having their trades fail, but also their positions exposed which can result in unfavorable price movements.

Current Exchange Protocols (e.g. IDEX, DDEX, 0x relayers, Airswap, Oasis DEX, etc.) all either use a centralized server or a trusted third party for order relay and matching. The only exceptions are EtherDelta which is a fully on-chain DEX that refreshes its orderbook every block on Ethereum and Paradigm Market which is a Tendermint-based relayer protocol on 0x. Paradigm Market is a newer proposal which outlines a decentralized relayer protocol but currently has no implementation at the time of writing. Paradigm's protocol uses a Tendermint blockchain that maintains a orderbook using BigchainDB and relies on nodes in its relayer network to relay orders and reach consensus on the orderbook for each state. Paradigm's design has several major security flaws including node front-running, between-state collision, and race-condition front-running. We consider Paradigm to be a progression in decentralizing relayers and preventing collisions, but it is fully vulnerable to front-running and therefore unsatisfactory.

Batch Trading was originally proposed[5] as a mechanism in market design to disincentivize predatory high frequency trading firms in traditional financial markets and was later integrated in decentralized exchanges by Hallex[6], Omisego[7], and Paradex to prevent front-running and collisions. In a batch traded exchange, relayers (or more generally, order matchers) have the sole power to group orders with the same price into a pool and fill multiple orders simultaneously at a single price, thereby resolving order collisions and removing a front-runner's ability to profit from order manipulation. Batch trading is a useful solution that allows for efficient liquidity aggregation and order matching within a liquidity pool, as it allows for fewer communication rounds between traders to fill orders. However, traders must trust their order matcher to maintain a fair orderbook and not front-run or censor their orders. As such, batch trading is neither truly *decentralized* nor

trustless. Current decentralized exchanges (such as Omisego[7]) implementing batch trading on 0x further have the weakness of limiting trading to a closed liquidity pool which thus prevents traders from accessing greater pools of liquidity from other relayers.

The concept of a ***Trade Execution Coordinator*** (TEC) was introduced to regulate trades and make the order matching process more fair. TECs eliminate the threat of front-running and trade collisions by approving trades and requiring that new trades do not conflict with an existing approved order. Trade settlement then follows price-time priority. The role of TEC can be played by a centralized, trusted entity or a group of federated stakeholders. Both do not fully satisfy *trustlessness* and *decentralization* since traders must trust the TEC to act honestly. In practice, federated stakeholders are also vulnerable to bribery from predatory front-runners as the profit from bribery can exceed the cost of an inefficient exchange. One potential trustless TEC proposal[4] specifies hashing the order information first and then the TEC can simply approve the transaction if the hash was not previously approved. However, this model is susceptible to a grinding attack where a dishonest TEC reconstructs the trade hash through brute force computation. Currently the concept of TEC is still a theoretical proposal with no implementations. We find that decentralized TECs to be a potentially useful mechanism to prevent collisions and front-running if we integrate our secure, trustless, and decentralized protocol which resolves the aforementioned vulnerability.

Commit-Reveal[4] is a commitment scheme that can also be used to prevent front-running while maintaining the desired properties of decentralization and trustlessness. The commit-reveal scheme is fairly simple and requires no trusted-third party: a taker first submits an intention to perform some trade (i.e. filling some make order) in the form of a hashed message that contains the trade information to the DEX. Then after at least one block is mined, the taker reveals the trade information to the DEX which in turn verifies the validity of the trade by reconstructing the hash. Although this setup does not resolve accidental collisions, it prevents front-running from other actors since trade information is encrypted. We find that this setup is optimal for completely preventing front-running, but also recognize that it creates a suboptimal user experience since the order confirmation and settlement require two transactions in different blocks.

In summary, we see that there are multiple proposed solutions to resolve the collision and front-running issues that DEXs face. However, there is currently yet to be an approach that does not sacrifice *decentralization*, *trustlessness*, *liquidity*, and *user experience*.

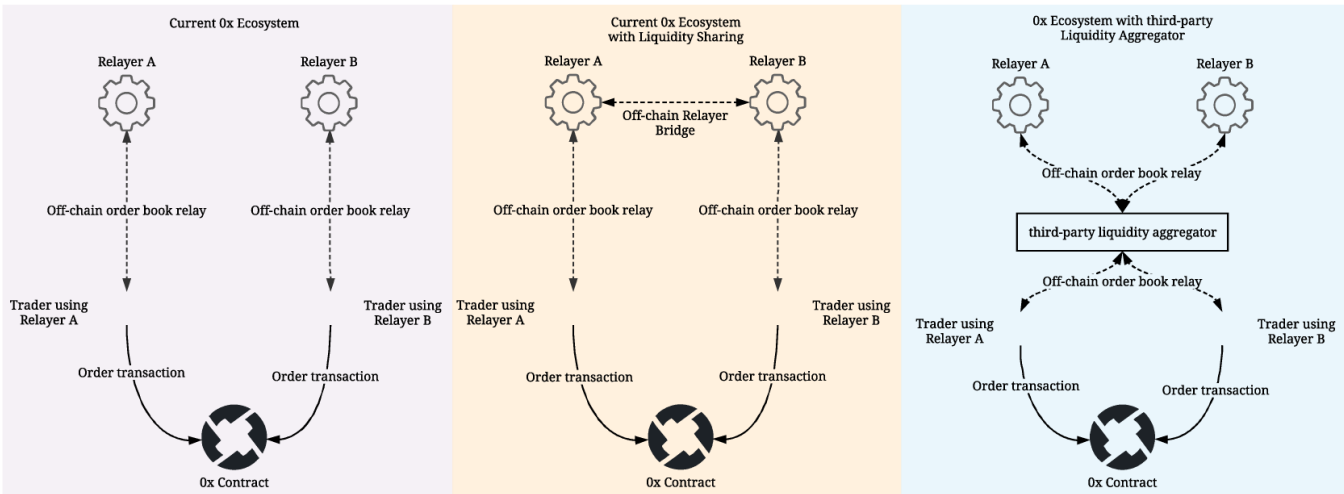
2.3 Liquidity Aggregation

In order for DEXs to match the user experience of centralized exchanges, orderbooks should have high amounts of liquidity. In practice, the amount of liquidity that DEXs possess pales in comparison to centralized exchanges. Efforts to increase liquidity in DEXs are only further hampered by the fact that many DEXs can only access liquidity from their own pool of traders. To solve this issue, some relayers of DEXs (e.g. OmegaOne, Enclave, and Totle) have coordinated with each other to share liquidity pools through ***liquidity aggregation*** in order to increase their liquidity. In liquidity aggregation, an aggregator coordinates with multiple relayers to relay orders from multiple orderbooks to traders and funnel trades to multiple DEXs. From a trader's perspective, a liquidity aggregator DEX will appear to have many more make orders since its orderbook is a combination of multiple exchanges.

Liquidity aggregation is a natural and elegant solution for combining sharded liquidity and we view it as essential to enable the DEX ecosystem to scale. However, current liquidity aggregators have various degrees of centralization and there is limited development for trustless and fully decentralized liquidity aggregators. Currently, traders must trust aggregators to be honest in displaying the true state of the aggregated orderbook. A dishonest aggregator could favor certain relayers' liquidity pools (e.g. collusion), censor some orders, or even front-run the traders themselves.

Since liquidity aggregators are centralized by nature, it is therefore essential to ensure that aggregators do not have the incentive to act dishonestly. Yet, in the scenario where individual relayers

also act as liquidity aggregators by using liquidity bridges, there is a perverse incentive for relayers to act dishonestly when orders collide. In the diagram under the Current 0x Ecosystem with Liquidity Sharing section below, a trader (shown in green) interacting with Relayer A wishes to fill a take order from Relayer B's liquidity pool and submits an order to Relayer A to do so. Relayer A then communicates this order with Relayer B through the Relayer Bridge. Later but within the same block period, suppose a trader (shown in red) from Relayer B's liquidity pool submits a conflicting take order to Relayer B. Assuming that relayer fees are split whenever a relayer bridge is used (which is standard practice), Relayer B has an incentive to prioritize the order from the trader from its own liquidity pool, as doing so will allow Relayer B to receive a larger amount of fees.



Even assuming that relayers act honestly, there can still be order collisions from conflicting take orders submitted by traders in the same block. As mentioned in the beginning of this section, the same race condition occurs which results in trades with the highest gas fee taking precedence. The underlying reason for this problem is that one cannot fairly and securely determine who should be the winning taker, as submitted timestamps are unreliable in an asynchronous network. This root problem motivates the existence of a better protocol which overcomes such limitations and enables a truly fair and trustless decentralized exchange.

3 Injective Protocol Settlement Logic

In order to resolve the aforementioned issues of front-running and collisions while still enabling sharing of liquidity in a fully decentralized and trustless way, we introduce Injective Protocol. Injective Protocol is a novel on-chain settlement logic scheme that establishes a fair sequence of incoming orders, thereby resolving collisions and front-running. Injective Protocol accomplishes this by using a publicly verifiable *proof-of-elapsed-time* construction using *verifiable delay functions*[2] which also allows for seamless liquidity sharing between relayers. This settlement logic is also protocol-agnostic, enabling multiple DEX protocols to settle trades and share liquidity in a fully decentralized and trustless fashion.

3.1 Preliminary Definitions

Definition 3.1. A market consists of *Make Orders* and *Take Orders* :

- M is the *make order* and M_{vol} is the volume of the *make order*.

- T is the *take order* sent by *taker* T_a with address A. T_b is a conflicting or adversarial *taker* with address B who submits a *take order* for the same *make order* M as T_a's order. We also define T_{vol} as the volume of this take order.

Definition 3.2. A simplified implementation of a *verifiable delay function* (VDF) is a tuple of algorithms where $V = \mathsf{Setup}(\lambda, t), \mathsf{Eval}(ek, x), \mathsf{Verify}(vk, x, y, \pi)$

- $\mathsf{Setup}(\lambda) \Rightarrow \mathbf{pp} = (\mathsf{ek}, \mathsf{vk})$ takes a security parameter λ and generates public parameter $\mathbf{pp}$ that consists an evaluation key ek and a verification key vk. In practice $\mathbf{pp}$ will remain static at the launch of the network. We simplify our definition of VDF by omitting puzzle difficulty parameter t which was included in the original definition [2] of VDF, since t is incremental in our protocol and will be submitted as part of the proof.

- $\mathsf{Eval}(ek, x) \Rightarrow (\mathsf{y}, \pi)$ takes an input x and generates y and proof π. In practice Eval is a deterministic, parallelize-hard function that has an exponential gap between evaluation and verification.

- $\mathsf{Verify}(vk, x, y, \pi) \Rightarrow \{True, False\}$ is a deterministic verification algorithm that takes the input x, output y, and outputs a boolean statement True or False. In practice (especially for VDF candidates that do not satisfy *decodable* property) the algorithm will also evaluate proof π.

Definition 3.3. *Taker* computes VDF locally and continuously submits the following to the settlement logic smart contract

$$\mathsf{Commit}(\pi, t, x, y_t, vk, T, S_r)$$

where π is a list of proof generated from the verification algorithm (for simplicity, we abbreviate $\{\pi_t | t \in \mathbb{N}\}$ as π), t is the number of iterations of Eval that the *taker* has computed, x is the starting input encoding the take order trade information T, y_t is the output of the t^{th} iteration of Eval, T is the *take order* the *taker* desires, and $S_r = \mathsf{Sig}(\mathsf{sk}, m = nonce_{r-1}||T))$ is the *taker*'s signature with his private key sk over the nonce of previous block concatenated with the take order.

Definition 3.4. The settlement logic receives Commit and evaluates it with

$$\mathsf{Confirm}(\pi, t, x, y, vk, T, S_r)$$

which verifies whether T_a's Commit is valid using $\mathit{Verify}(vk, x, y, \pi) \Rightarrow \{True, False\}$. Suppose within a given block, there are n $\mathsf{Commits}$ which signal the desire to take a given make order M. When the settlement logic receives more than one Commit for a *make order*, it will select the Commit with the t_n corresponding to $\max\{t_n | n \in \mathbb{N}\}$ from the t's obtained from each of the n conflicting Commit functions as a temporary winner of *make order* M. We also define δ as the delay in number of rounds (blocks) between the first temporary winner and the resolution of permanent winner. δ can be parameterized for decentralized exchanges as desired and is discussed in greater detail in the next section.

Definition 3.5. In the most simple case, assuming $M_{vol} < T_{a(vol)} + T_{b(vol)}$ where the combined *take orders* have a higher volume than the *make order* volume, a *collision* occurs when T_a and T_b both submit a *take order* for the same M within the same round r and the settlement logic cannot deterministically select the winning *taker* in a fair fashion. *Front-Running* occurs when T_a submits a valid *take order* first but an adversarial T_b submits the same *take order* for M in an attempt to take M before T_a is confirmed (typically done by simply using a higher gas fee), even though T_b was sent after T_a.

3.2 Verifiable Delay Functions as Proof of Elapsed Time

We utilize *verifiable delay functions* (VDF) as a mechanism for *proof of elapsed time* where a *taker* can submit a time proof to the settlement logic which verifies whether the *taker* has indeed observed the order for the duration he claims from Commit. Beyond satisfying *sequential* and *efficiently verifiable* properties proposed in VDF[2], we also require our VDF candidates to be *incremental*, allowing hardness parameter t to be specified in the proof instead of in Setup.

For the *taker* submitting take order T at a given block r, we denote the taker's starting value x as:

$$x = H(\mathsf{Sig}(\mathsf{sk}, m))$$

where $m = nonce_{r-1}||T$ (i.e. the nonce of the previous block concatenated with the take order), sk is T_a's secret key, and $\mathsf{Sig}(\mathsf{sk}, m)$ is a digital signature on message m. For simplicity, we denote $S_r = \mathsf{Sig}(\mathsf{sk}, m)$. This construction ensures that at a given block r, one cannot precompute x for future blocks since x depends on the nonce of the previous block $r-1$.

For Injective Protocol's settlement logic, we use four candidate VDF algorithms for Eval and Verify which are the following: Sloth++[2], Permutation Polynomial Chain on an injective rational map[2], Wesolowski's Efficient VDF[8], and Pietrzak's Simple VDF[9]. In Sloth++ or Permutation Polynomial Chain with SNARK proposed by Boneh, Bonneau, Bünz, Fisch, an *Iterated Sequential Function* in the form of $f(t, x) = g^{(t)}(x) = \underbrace{g \circ g \circ ... \circ g}_{t\ times}$ is used. Replacing g with our Eval algorithm and integrating Verify, we have:

$$\textit{computed in parallel} \begin{cases} x = x_0 \rightarrow \underbrace{\mathit{Eval}(x_0)}_{x_1} \rightarrow \underbrace{\mathit{Eval}(\mathit{Eval}(x_0))}_{x_2} \rightarrow ... \rightarrow \underbrace{\mathit{Eval}^{(t)}(x_0)}_{x_t} = y_{t+1} \\ x = x_0 = \underbrace{\mathit{Verify}(x_1) \leftarrow x_1}_{\pi_1} = \underbrace{\mathit{Verify}(x_2) \leftarrow x_2}_{\pi_2} \leftarrow ... \leftarrow \underbrace{\mathit{Verify}(x_t) \leftarrow x_t}_{\pi_t} = y_{t+1} \end{cases}$$

For Sloth++[2], a proposed improvement on the Sloth [10] function that integrates SNARK for succinct verification, given $x \in \mathbb{Z}_p^*$ where p is a sufficiently large prime such that $p \equiv 3 \bmod 4$, $\mathsf{Eval}(x) = \sqrt{x} \bmod p = x^{\frac{p+1}{4}} \bmod p$. For verification, $x \bmod p = \mathsf{Verify}(y) = y^2$, it will also produce $\mathit{SNARKProve}(\mathit{ek}, y) \Rightarrow \pi$.

For permutation polynomial chain on injective rational maps[2], we use a candidate polynomial proposed by Guralnick and Muller[11] over $\mathbb{F}_{p^m}$:

$$\frac{(x^s - ax - a) \cdot (x^s - ax + a)^s + ((x^s - ax + a)^2 + 4a^2x)^{(s+1)/2}}{2x^s}$$

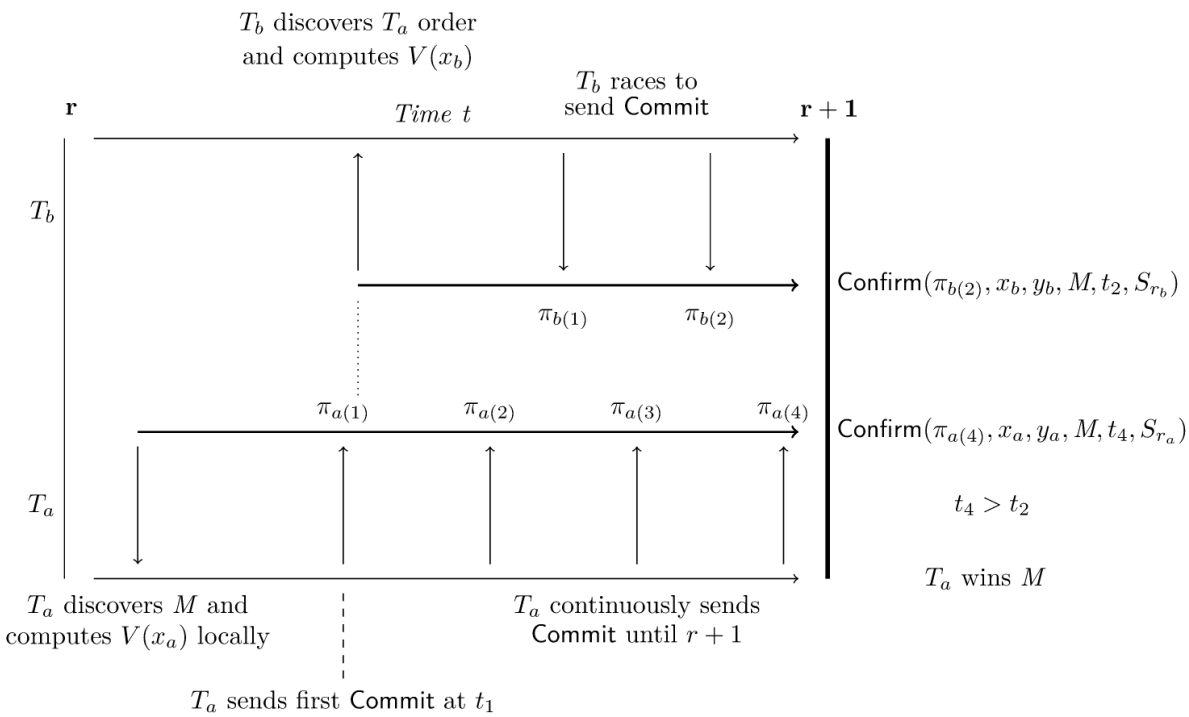
where $s = p^r$ for odd prime p and some integer r and a is not a $(s-1)$th power in $\mathbb{F}_{p^m}$. $\mathit{Eval}(x)$ will be an algorithm that calculates the inversion of the polynomial with difficulty parameter s. This proposal assumes that computing polynomial GCDs is the fastest inversion method, requiring at least s sequential steps even if evaluated on optimized hardware with at least s parallel processors. However non-optimized hardware with minimum parallel processors will evaluate Eval at $O(s^2)$ time.

Modifying Setup, we can implement Wesolowski's[8] construction that uses the Rivest, Shamir, and Wagner (RSW)[12] time-lock puzzle based on a trapdoor of group G of unknown order. We do not need an *Iterated Sequential Function* for an RSW construction as the difficulty parameter t is incremented sequentially during the evaluation. But this construction does not satisfy the *decodable* property that the previous two constructions do, as there is no efficient way of computing the inversion of the RSW algorithm. RSW assumes that given $y = x^{2^t} \bmod N$, an evaluator would need t sequential squarings to evaluate y if the group order or factorization of N is unknown. To implement this, we will modify Setup by creating an unknown group order using Wesolowski's[8] imaginary quadratic order. This way, Setup can simply choose a random discriminant, assuming that class group cannot be computed faster than solving RSW when the discriminant is large.

Pietrzak[9] proposed a VDF candidate that improved upon Wesolowski's RSW puzzle by implementing a halving protocol, which allows for parallelism in the proof construction and a reduced proof time of $\approx\sqrt{t}$ time. However, this comes at a cost of proof size and verification time at a factor of $log(t)$.

3.3 1-Round Settlement Model

We propose a model where orders are settled in the same round that they are made ($\delta = 0$) where the settlement logic deterministically selects the *take order* with the highest t. From round r to $r+1$, *takers* continuously submit a sequence of Commits with incremental t until the end of the round and the settlement smart contract will evaluate the winning *take order* at round *r+1*.



In the above diagram we have T_a, an honest *taker* with his starting value x_a, and T_b an adversarial front-runner (or a colliding honest *taker* who submitted for the same *make order* M but at a later time) with his respective starting value x_b. T_a computes $V(x_a)$ locally and submits the first proof $\pi_{a(1)}$ with some delay t_1, which he chooses. T_b, after observing T_a's first Commit will race to compute t. However, due to T_a's head-start, it would be exponentially more difficult for T_b to generate a final t that's larger than T_a's, even if T_a is using non-optimized hardware.

However, this model is only collision-proof and front-running *resistant*. Although doing so would be economically costly and risky, an adversary could precompute all *make orders* in an orderbook $\{M_1, M_2, ..., M_n\}$ in parallel at the start of each round and submit a final t higher than that of an honest T_a after it observes T_a's first Commit (grinding attack). Network nodes or miners could also block T_a's follow up Commit transactions and then submit their own Commits with a higher t value, thereby successfully front-running T_a's order.

Still, this basic model is useful for settling inter-exchange trades and offering collision resistance for closed-liquidity relayers as well as for relayers that already have front-running resistant measures.

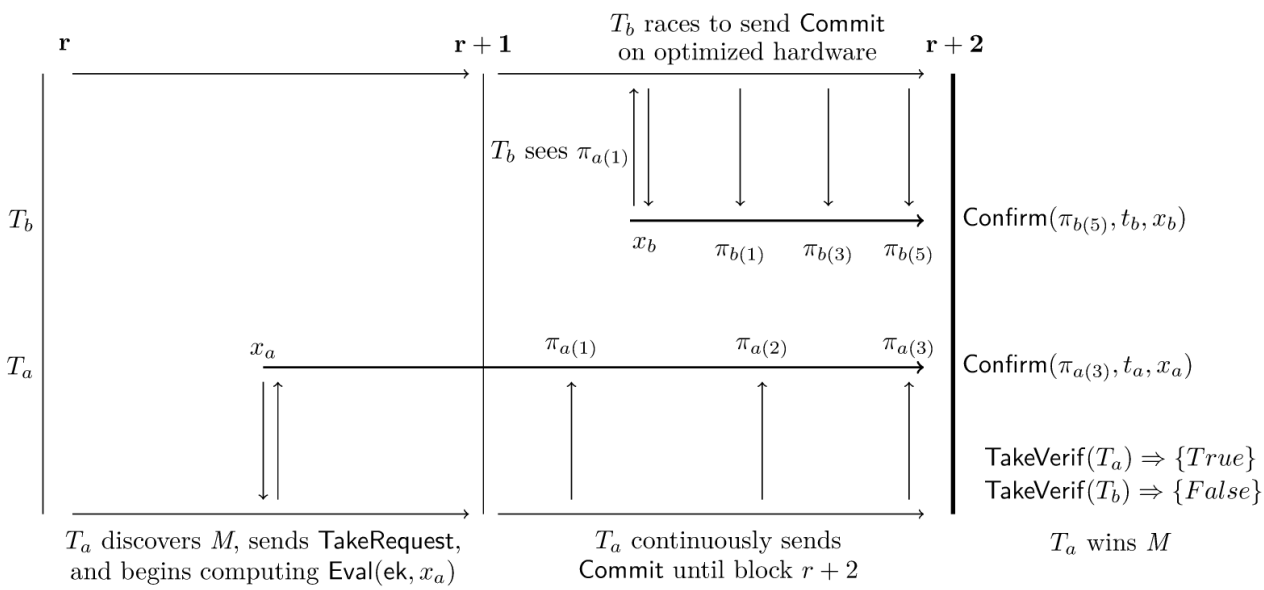
3.4 Front-Running Proof Model

To fully combat front-running, we propose a model that implements a commit-and-reveal scheme[4] with a delay factor of $\delta \geq 1$.

Definition 3.6. Recall that for a *take order* T submitted to the settlement smart contract by T_a at block r, $x_0 = H(\mathsf{Sig}(\mathsf{sk}, m = nonce_{r-1}||T)) = H(S_r)$ as previously defined. T_a submits x_0 (denoted as x_a) to validate his T (i.e. "revealing" it) by sending the corresponding Commit after at least 1 round but within δ rounds. The *taker* then uses x_a as the starting value for his VDF construction. To prevent a *grinding attack* where adversaries attempt to reconstruct the hash by guessing all combinations, *commitment* information x_a is comprised of T_a's signature on a message $nonce_{r-1}$ and T.

Definition 3.7. T_a submits his intent to take T by sending a $\mathsf{TakeRequest}(x_a, A)$ transaction to the settlement logic. At a later block within the δ block time frame, T_a will submit Commits which will be verified with TakeVerif where $\mathsf{TakeVerif}(H(S_r), x_a) \Rightarrow \{True, False\}$.

In the below diagram, we have a model with $\delta = 1$ and T_b is an adversarial front-runner attempting to front-run T_a's order:



In this scenario, T_a sends TakeRequest at block r and submits Commits during block $r + 1$. T_b seeing T_a's Commits, discovers T and attempts to front-run the trade by submitting Commits with a higher hardness parameter t using optimized hardware. Since T_b did not send TakeRequest in block r, its *take order* is deemed invalid by *TakeVerif* and was rejected, even though its t was higher.

In order to still have the opportunity to front-run, T_b can submit TakeRequests for every M on the orderbook for every single round and simply not submit Commits so the orders are never revealed and hence never filled. Beyond the fact that doing so would incur significant gas fees, this attack can be made even more costly by requiring the trader to post a deposit which will be forfeited if a large number of TakeRequests are submitted without a follow up Commit.

3.5 Practical Considerations

In the scenario where two takers submit the same TakeRequest at different times but still within the same block, a fair system would allow the taker who submitted the TakeRequest first to win the trade. Because obtaining an accurate timestamp for a given TakeRequest is impossible in an asynchronous network, we use the t value in Commit to adjudicate who submitted the TakeRequest first in the previous block. However, this system allows a trader using optimized hardware would be able to win traders over a trader using commodity hardware even if the trader using commodity hardware submitted the TakeRequest earlier.

We first mitigate this issue by selecting VDF candidates that offer no advantage when computed on parallel processors. Sloth++, Pietrzak's RSW VDF, and Wesolowski's RSW VDF are three candidates that satisfy this property due to their sequential nature. However, it is still unclear which VDF candidate is best suited for our setup. In Wesolowski's VDF and Pietrzak's RSW, proof generation occurs after VDF evaluation while in Sloth++, proof generation can be computed in parallel. Furthermore, proof generation takes $O(t)$ time for Wesolowski's RSW but only $O(\sqrt{t})$ time for Pietrzak's RSW. However, this speedup comes at a sacrifice of longer verification time and larger proof size by a factor of $\log(t)$. Extensive testing will be required to determine the best VDF candidate for our use case.

In order to equal the playing field, traders could delegate the VDF computation and proof generation to external computation providers that run on optimized hardware. However, we recognize that doing so introduces elements of centralization and can create a suboptimal user experience.

3.6 Sidechain Implementation

In our setup, the settlement logic is executed by a trade execution coordinator (TEC) which uses the 0x V2 protocol[13]. The smart contract address of the TEC will become the `senderAddress` parameter for any *make order*, which will enforce that only *take orders* approved by the TEC will successfully fill the *make order*. In turn, the TEC smart contract establishes the true sequence of incoming take orders by using a sidechain which uses the Front-Running Proof Model described in the previous section. The TEC then allows the *take orders* it finds valid to succeed by performing the exchange on 0x. By using a sidechain for trade execution, traders can submit Commits to the sidechain without consuming significant gas. Besides saving on gas, another advantage is that the sidechain settlement logic is protocol agnostic, meaning it can determine a fair sequence of orders for any DEX protocol.

Furthermore, the TEC on the sidechain can allow for liquidity sharing across several distinct DEXs. For example, the TEC could relay orders from another decentralized exchange such as IDEX [14], thus creating shared liquidity. As described in 2.3, the TEC can also serve as a liquidity aggregator, allowing a trader to access and trade on multiple orderbooks from multiple exchanges.

4 Trustless Relayer Network

In conjunction with our settlement logic proposal, we introduce Injective Relayer - a decentralized, trustless relayer network hosted on the same sidechain. The relayers in this network match orders using a non-interactive commit-reveal scheme which prevents trade censorship and front-running. The relayers are also trustless insofar as individual relayers have no incentive to act dishonestly.

The relayer network is hosted on a sidechain that satisfies the following properties (assuming less than 1/3 of nodes are Byzantine):

- *Decentralized*: The network is permissionless and allows any participant to join the network as validators with minimal barriers.

- *Frequent State Machine*: The network updates in a stateful nature.
- *Trustless*: An individual node cannot profit from front-running any incoming trades or manipulating the sequence of orders.
- *Consistent*: The orderbook is always accurate and forks are prevented (consistency is prioritized over availability under the CAP theorem [15])

4.1 Preliminary Definitions

Definition 4.1. Extending our VDF construction from 3.2, we use Encrypt as component of a *time-lock puzzle* as follows:

- $\mathsf{Encrypt}(y, \mathsf{vk}) \Rightarrow x$ which takes y and verification key vk and outputs starting value x which is used as the input for Eval.

Definition 4.2. The *taker* submits an encrypted *take order* $E(T)$ containing his order information which is comprised of his address A, price P, volume V, *Maker* address M, and *taker* signature S_r at round r:

$$E(T) = E(\mathsf{A}||P||V||\mathsf{M}||S_r, k)$$

where a *taker* takes his *take order* message $m = T = \mathsf{A}||P||V||\mathsf{M}||S_r$ and encryption key k and performs symmetric encryption $E(T)$ to generate the encrypted order information. In our construction, k is generated from T. Therefore for simplicity, we abbreviate $E(T, k)$ as $E(T)$.

Definition 4.3. The relayer orderbook is hosted on the sidechain as a stateful system with an inconsistent state transition time. We denote σ as a given state, M_σ as an orderbook state which consists of all outstanding *make orders*, T_σ as a list of all the decrypted *take orders* in a state, and function Match: $M_{\sigma_N} - T_{\sigma_N} \Rightarrow M_{\sigma_{N+1}}$ where N is the current state number. Match is an order verification and matching algorithm such that for every *take order* T, it verifies the signature S_r, performs a balance check, prunes the orderbook, executes the trades, and updates the individual *make order*'s state.

4.2 Verifiable Delay Functions as a Time-Lock Puzzle

We construct a *time-lock puzzle* using *verifiable delay function* candidates. For a naive construction we require that the candidates satisfy *decodable, sequential,* and *efficiently verifiable* properties. Hence, permutation polynomial chain and Sloth++ satisfy the aforementioned properties while candidates derived from RSW do not.

For Sloth++ and permutation polynomial chain, our proposed Encrypt algorithm is the inversion of the Eval function. In Sloth++'s case, given $y \in \mathbb{Z}_p^*$ and $p \equiv 3 \bmod 4$, Encrypt will be $\mathsf{Encrypt}(y) = y^{2^t} \bmod p = x$. Therefore Eval requires t iterations of modular square roots to retrieve y. Encrypt must inform whether message y is a quadratic residue as $\sqrt{x} \bmod p$ yields both $y \bmod p$ as well as $-y \bmod p$. Hence, knowing this property allows the relayer nodes to distinguish the intended message from y and $-y$. However, this information allows attackers to limit the prime field p by half in an ideal p where half of the field's integers are quadratic residues.

Using a permutation polynomial chain does not exhibit this reduction of the prime field but requires more parallel computation resources to compute the puzzles efficiently. Computing a single iteration of a candidate polynomial with degree d takes a non-parallel processor $O(d^2)$ time while the same would take a parallel processor only $O(d)$ time. This property creates scalability obstacles for relayer nodes since computations required for decrypting incoming orders can exceed a relayer's available parallel processing power, causing delays in orderbook update. However, this weakness can be mitigated with proper fault-tolerant task scheduling.

The relayer network uses a symmetric-key encryption algorithm AES [16] that allows a *taker* to send a ciphertext along with a key encrypted by backward VDF function. When a *taker* wishes to submit an encrypted order, he creates a key with the hash of take order information $H(T) = k$, generates starting value x for Eval with $\mathsf{Encrypt}(y = H(T), \mathsf{vk}) \rightarrow x$, and then uses k to encrypt the full *take order* information $E(T, k)$ with AES. We simplify $E(T, k)$ as $E(T)$ as the key k is generated from taker information T. After generating $E(T)$, the *taker* then submits $E(T)$ along with x to the relayer nodes:

$$\mathsf{Submit}(E(T), x)$$

A relayer node can decrypt the cipher $E(T)$ by computing AES decryption of the ciphertext $E(T)$ along with the decryption key k which can obtained by computing the forward VDF on x with Eval.

$$\mathsf{Eval}(\mathsf{ek}, x) \Rightarrow H(T) = k$$
$$D(E(T), k) = T$$

To verify that the forward VDF computation was done properly, other relayers can check the proofs π obtained from the Eval computation. When a decrypted take order information is indecipherable or incorrect, the proof π can be used to identify whether a relayer node evaluated the VDF improperly or the *taker* submitted an invalid order.

4.3 Non-Interactive Commit-Reveal Order Matching

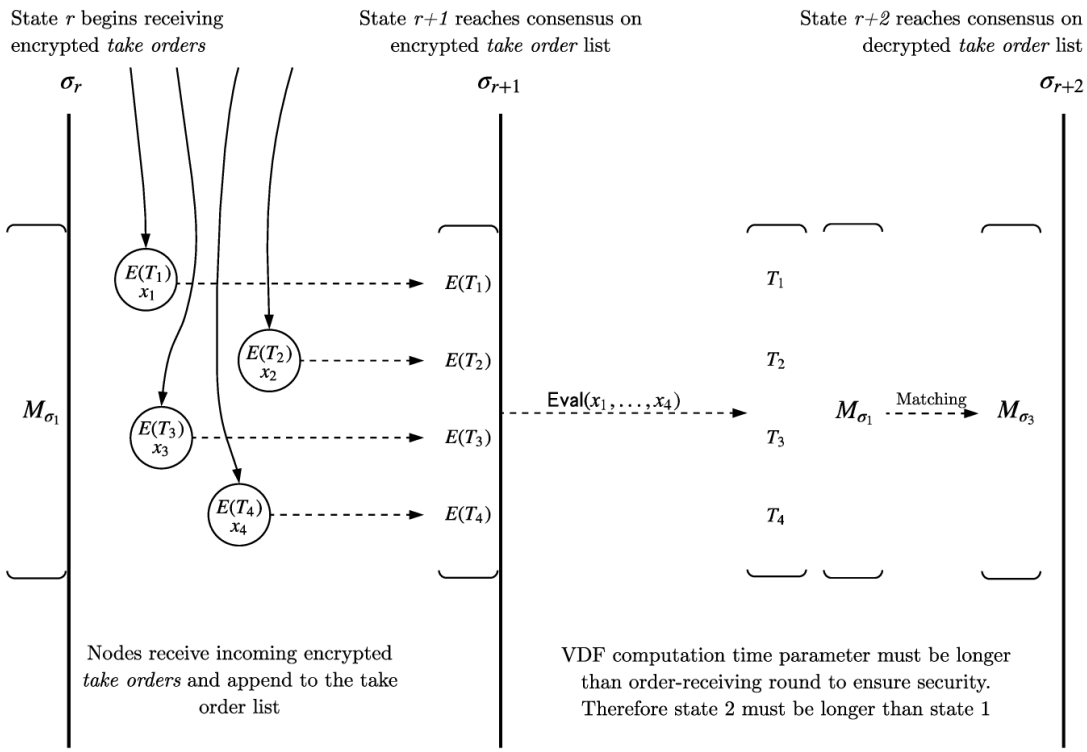
To prevent front-running, we implement a commit-reveal scheme where a *taker* first submits the encrypted order information and the information is revealed at a subsequent state. In the relayer network, we use *time-lock puzzles* to make the commit-reveal scheme non-interactive, allowing traders to make trades with only one interaction with the relayer network. As a result, there is no opportunity for malicious relayers to prevent traders from rejecting a *taker's* transaction that includes the revealed order information since no such transaction exists. The network is structured as follows:

The relayer network updates the orderbook every 2 states. In state r, denoted as σ_r, the relayer nodes aggregate incoming Submit transactions, which each includes encrypted take order information $E(T)$ and starting value x, from *takers* into a list of encrypted take orders $\{(E(T_\alpha), x_\alpha) | \alpha \in \mathbb{N}\}$ where the total number of orders in the list is denoted as α.

After reaching consensus on the take order list at the end of state $r+1$, the relayers decrypt each take order in the list with $D(E(T), k)$. To prevent the relayers from front-running the orders by decrypting $E(T)$ received in state r before state $r+1$ begins, the VDF evaluation time (denoted as t_{VDF}) must be $t_{\sigma_r} < t_{VDF} < 2t_{\sigma_r}$. Since t_{VDF} is part of $t_{\sigma_{r+1}}$, this implies that $t_{\sigma_r} < t_{\sigma_{r+1}} < 2t_{\sigma_r}$. The hardness parameter t can be periodically adjusted so t_{VDF} can be maintained.

After a relayer node successfully decrypts the take order list, it can submit the list along with the proofs to the network. The proofs will be then verified on a smart contract maintained by the network. The first relayer node that decrypts the list will in turn earn some reward. The network then matches the orders deterministically and reaches a consensus on the updated orderbook $M_{\sigma_{r+2}}$ at state $r+2$.

The caveat of this model is that the orderbook updates every 2 states while communication with the *takers* only occur in 1 of them. As a result, a trader with an unstable connection could submit *take orders* based off of a stale orderbook, thus resulting in order failure. It is also important to note that the network cannot include $E(T)$ from expired states into M_σ, as a front-running relayer node could easily attack the network by delaying all incoming $E(T)$ until it decrypts them and submitting them in subsequent states.



4.4 Sidechain Attack Vectors

- A **Sybil Attack** occurs when a malicious trader floods the network mempool with false encrypted orders that contains either invalid order information or simply random ciphertexts. Since the relayer cannot determine the validity of an order from the ciphertext, computation is wasted during the decryption process. We can prevent this attack by implementing a staking mechanism that forces traders to stake a fixed amount of token prior to making trades. If the network finds that a trader intentionally submits a falsely encrypted order, the trader's stake is forfeited to compensate the relayer nodes for the wasted computation. However, this design clearly results in a suboptimal user experience for traders, as they have to make a transaction to purchase the token to stake prior to trading. To resolve this issue, we propose a more flexible model where traders have the option to send encrypted or unencrypted orders. Traders who send unencrypted orders would not have to stake tokens but would pay higher exchange fees and accept the risk of getting front-run. This mechanism should incentivize traders making large trades to opt for submitting encrypted orders and staking the native tokens.

- A **33% Attack** occurs when the nodes with more than 1/3 of the voting power are Byzantine. When this occurs, since the network requires at least 2/3 of the voting power to reach consensus, the network will halt. However, since funds are never deposited to the sidechain, users will never experience loss of funds.

References

[1] MtGox. Announcement regarding the applicability of us bankruptcy code chapter 15, 2014. https://www.mtgox.com/img/pdf/20140314-announcement_chapter15.pdf.

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

[3] Will Warren and Amir Bandeali. 0x: An open protocol for decentralized exchange on the ethereum blockchain. *URl: https://github. com/0xProject/whitepaper*, 2017.

[4] Will Warren. Front-running, griefing and the perils of virtual settlement (part 2). Medium, Feb 2018. https://blog.0xproject.com/front-running-griefing-and-the-perils-of-virtual-settlement-part-2-921b00109e21.

[5] Eric Budish, Peter Cramton, and John Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response *. *The Quarterly Journal of Economics*, 130(4):1547–1621, 2015.

[6] Sam Fisher Jakob Hautop Nicolai Larsen Jeppe Hallgren, Malte Hallgren and Omri Ross. Hallex: A trust-less exchange system for digital assets, 2017. https://hallex.com/hallex.pdf.

[7] OmiseGO Team Joseph Poon. Omisego: Decentralized exchange and payments platform, 2017. https://cdn.omise.co/omg/whitepaper.pdf.

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

[9] Krzysztof Pietrzak. Simple verifiable delay functions. Cryptology ePrint Archive, Report 2018/627, 2018. https://eprint.iacr.org/2018/627.

[10] Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. Cryptology ePrint Archive, Report 2015/366, 2015. https://eprint.iacr.org/2015/366.

[11] Robert M. Guralnick and Peter Müller. Exceptional polynomials of affine type. *Journal of Algebra*, 194(2):429 – 454, 1997.

[12] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.

[13] 0x team. 0x protocol 2.0.0 specification. Github, Sep 2018. https://github.com/0xProject/0x-protocol-specification/blob/master/v2/v2-specification.md#creating-an-order.

[14] Aurora Labs. Idex: A real-time and high-throughput ethereum smart contract exchange, 2017. https://idex.market/static/IDEX-Whitepaper-V0.7.5.pdf.

[15] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[16] Vincent Rijmen Joan Daemen. Aes proposal: Rijndael, 1999. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf.