arXiv:2204.00955v4 [cs.CR] 15 May 2025

FIRST: FrontrunnIng Resistant Smart ConTracts

Emrah Sariboz, Gaurav Panwar, Roopa Vishwanathan, Satyajayant Misra
{emrah,gpanwar,roopav,misra}@nmsu.edu
New Mexico State University
Las Cruces, NM, USA

ABSTRACT

Owing to the increasing acceptance of cryptocurrencies, there has been widespread adoption of traditional financial applications such as lending, borrowing, margin trading, and more, into the cryptocurrency realm. In some cases, the inherently transparent and unregulated nature of cryptocurrencies exposes users of these applications to attacks. One such attack is *frontrunning*, where a malicious entity leverages the knowledge of currently unprocessed financial transactions and attempts to get its own transaction(s) executed ahead of the unprocessed ones. The consequences of this can be financial loss, inaccurate transactions, and even exposure to more attacks. We propose FIRST, a framework that prevents frontrunning, and as a secondary effect, also backrunning and sandwich attacks. FIRST is built using cryptographic protocols including verifiable delay functions and aggregate signatures. We formally prove the security of FIRST using the universal composability framework, and experimentally demonstrate its effectiveness using Ethereum and Binance Smart Chain blockchain data. We show that with FIRST, the probability of frontrunning is approximately 0.00004 (or 0.004%) on Ethereum and 0% on Binance Smart Chain, making it effectively near zero.

KEYWORDS

Frontrunning Prevention, Smart Contract Security, MEV Mitigation, Transaction Ordering Fairness.

ACM Reference Format:
Emrah Sariboz, Gaurav Panwar, Roopa Vishwanathan, Satyajayant Misra. 2025. FIRST: FrontrunnIng Resistant Smart ConTracts. In *Proceedings of ACM Conference (Conference'21).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The decentralized, trustless, and censor-resistant nature of Ethereum, along with its support for smart contracts, has enabled a wide range of financial applications and has created the Decentralized Finance (DeFi) ecosystem, which is worth more than 75 billion USD as of December 2024 [2]. With the recent developments, many real-world financial products such as money lending and borrowing, margin trading, exchange platforms, derivatives and more, are being made available to the blockchain users via smart contracts [1, 11, 38].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'21, July 2021, Virtual
© 2025 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

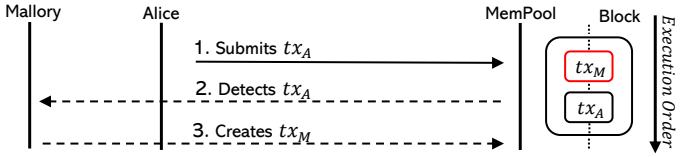


Figure 1: Steps involved in a frontrunning attack.

Unfortunately, the absence of regulations allows malicious actors to adopt and employ dubious practices from traditional finance within the cryptocurrency ecosystem.

In finance, frontrunning is an act of purchasing stock or other securities right before a large (whale) transaction owing to access to non-public information. By doing so, one can take advantage of the outcomes of large unprocessed transactions to be executed after a later time than one's own. Frontrunning has been classified as illegal by monitoring entities, such as the U.S. Securities and Exchange Commission (SEC) and principally prevented by extensive regulations [28]. In permissionless chains such as Ethereum, transactions that do not utilize private relayers like Flashbots [30] or directly interact with validators to obscure their details are publicly visible in the pending pool (or mempool) before being processed. This visibility allows adversaries, such as Mallory, to monitor the peer-to-peer (P2P) network for potentially exploitable transactions. For example, when an honest user, Alice, submits a transaction tx_A with a gas price of G_A, Mallory can craft a competing transaction tx_M with a higher gas price G_M, where $G_M > G_A$, ensuring tx_M is prioritized and included in the next block before tx_A. Figure 1 illustrates a typical frontrunning attack.

Examples of frontrunning attacks can be seen on various decentralized applications (dApps). The first and most prominent attack vector is on decentralized exchanges (DEXes). DEXes are exchange platforms built on smart contracts that enable users to exchange assets without the need for an intermediary [32]. Unlike centralized exchanges, where users wait for their buy/sell orders to be fulfilled, most DEXes—e.g., Uniswap [38]—use an automatic pricing mechanism known as an Automated Market Maker (AMM) to perform instant trades. A frontrunner can perform attacks with highly predictable results due to deterministic pricing mechanism as well as the transparency of liquidity amounts of decentralized exchanges. In this context, Qin *et al.* estimated a profit of 1.51 Million USD made by frontrunners [32]. Other domains that are affected by frontrunning attacks include (but are not limited to) gambling [22], bug bounty programs [16], smart contract exploits [37], and clogging [32], which emphasizes the threat and the need for mitigation. In this work, we aim to mitigate frontrunning attacks on blockchains that support smart contracts such as Ethereum, but without modifying the blockchain's underlying infrastructure. The core idea behind FIRST is to prevent Mallory from frontrunning Alice's transaction, tx_A, with her own transaction, tx_M, by ensuring

that tx_A is included in the block before tx_M. We introduce a novel approach to achieve this by leveraging Verifiable Delay Functions (VDFs) to impose a delay on tx_M [13]. Specifically, FIRST requires Mallory's transaction to wait for a predetermined amount of time, during which the VDF is evaluated, before it can interact with the dApp implementing the frontrunning protection. Once Mallory completes the VDF evaluation, she generates a proof that is then verified by a set of verifiers. Transactions that fail to be verified by the verifiers are rejected from calling the smart contract function.

Importantly, FIRST is not limited to a specific application and can be employed by various dApps, such as auctions, decentralized name services, NFT marketplaces, and others that are susceptible to frontrunning attacks. FIRST helps protect users from frontrunning and backrunning attacks, and consequently from sandwich attacks as well [44]. FIRST operates entirely at the application layer, requiring no changes to the underlying blockchain or its consensus protocol, thus requiring no changes to the blockchain framework. The specific FIRST transactions require an additional amount of verification by the validators (miners), which is incentivized by the framework to ensure the transactions are picked up to be verified and added to the chain.

Our novel **contributions** are as follows: **a)** We propose FrontrunnIng Resistant Smart ConTracts (FIRST), a general-purpose solution to the frontrunning problem using cryptographic protocols, such as VDFs and aggregate signatures [15]. FIRST significantly curtails frontrunning attacks in EVM-based blockchains while requiring no changes to the underlying blockchain infrastructure. As an application-agnostic solution, FIRST can be easily adopted by any dApp. **b)** We discuss the effectiveness of FIRST and experimentally evaluate it using Ethereum and Binance Smart Chain transaction data. **c)** We rigorously prove the security of FIRST using the Universal Composability (UC) framework.

Paper Organization: In Section 2, we provide a concise explanation of relevant preliminary concepts. Section 3 presents the system model and threat model. In Section 4, we detail the construction of FIRST and its constituent protocols. Section 5 offers a comprehensive security analysis of FIRST. In Section 6, we elaborate on the implementation and evaluation of our system. We discuss the design choices and limitations of FIRST in Section 7. We review related literature in Section 8 and conclude the paper in Section 9.

2 PRELIMINARIES

2.1 Ethereum and DeFi

Bitcoin demonstrated blockchain technology's potential by enabling direct transactions between untrusted parties without a central authority. Ethereum expanded on this by introducing *smart contracts*—self-executing programs on the blockchain that activate when predefined conditions are met. Developed using languages such as Solidity or Vyper, these transparent smart contracts have led to the creation of dApps. Finance-related dApps have enabled DeFi, which is an umbrella term that includes various financial products (such as flash loans, asset management services, decentralized derivatives, and insurance services) available to any user with an internet connection in a decentralized manner [3, 10]. It allows users to utilize financial products at any time. Additionally, DeFi products enable end-users to employ them in a non-custodial fashion, giving users complete control over their money, as opposed to traditional financial services based on a custodial model.

2.2 Cryptographic Preliminaries

Verifiable Delay Function: A Verifiable Delay Function (VDF) is a deterministic function $f : X \rightarrow Y$ requiring a fixed number of sequential steps, T, to compute, with efficient public verification [13]. While time-lock puzzles [21, 34] also enforce T sequential steps, they lack public verifiability and focus on encryption, making them unsuitable for applications like FIRST, where proof of elapsed time is essential. Recent VDF constructions by Pietrzak and Wesolowski [31, 42] address these limitations. We adopt Wesolowski's VDF [42] for its shorter proofs and faster verification. For a detailed comparison of these schemes, see [14], and for the formal definition, refer to Appendix A.

Aggregate Signatures: An aggregate signature scheme allows the aggregation of n distinct signatures from n users, each on a distinct message of their choice, into a single signature [15]. Moreover, it allows the aggregation to be done by any party among the n users, including a potentially malicious party. By verifying the aggregate signature, one can be convinced that n distinct users have signed n distinct messages, which have been collected into a single signature. FIRST utilizes this cryptographic primitive to aggregate the verification results of a VDF proof.

3 SYSTEM AND THREAT MODEL

3.1 System Model

Parties: In our system, there exist four main entities. 1) A smart contract *SC* that resides on the Ethereum blockchain. 2) Alice, who is a legitimate user interacting with *SC* by creating a transaction tx_A that is potentially vulnerable to frontrunning attacks. Alice is equipped with a verification/signing keypair (pk_A, sk_A). She evaluates a VDF instance, $\mathcal{V}$, given to her by a set of verifiers. 3) A set of verifiers $\mathbb{V}$ who generate and send the public parameters of the VDF, $\mathcal{V}$, to Alice and verify the evaluated $\mathcal{V}$ and its proof of correctness that Alice submits to them. A *coordinator* C, is an entity picked from the members of $\mathbb{V}$ by Alice to help aggregate their signatures into a single signature. 4) Validators, whose goal is to construct blocks and propose them to the network, validate potential blocks received from other nodes, and process transactions. Finally, dApp creator (*dAC*), who implements applications such as auctions, exchanges, bug bounty programs, and Initial Coin Offerings (ICOs) which are known to be targeted by frontrunning attacks.

3.2 Threat Model and Assumptions

Mallory: We assume Mallory is an adversary who is computationally bounded and economically rational. Mallory is observing the pending transaction pool for Alice's transaction, tx_A on the Ethereum network. Mallory will attempt a frontrunning attack as soon as she observes tx_A on the pending pool by paying a higher priority fee. We also take into account the case where more than one adversary attempts to frontrun tx_A. For ease of exposition, we use Mallory to represent a group of adversaries.

Verifiers: Verifiers $\mathbb{V}$ are a set of entities not controlled nor owned

by the *dAC*. The protocol in its current version relies on the assumption of an honest majority among verifiers to guarantee the system's proper functionality. The trust assumption in the verifiers ensures that transactions are not subjected to unnecessary delays from the malicious verifiers, thereby maintaining the liveness property. In FIRST, verifiers do not have access to client details during $\mathcal{V}$ verification, effectively precluding transaction censorship.

While we acknowledge the trust assumption in this version, FIRST is designed with an intuitive plug-and-play framework that seamlessly integrates projects like Eigenlayer. This integration aims to minimize trust dependencies and align with Ethereum's renowned fault tolerance [36]. Eigenlayer offers Ethereum validators the opportunity to restake their ETH, thereby extending Ethereum's security to additional protocols. Just as with Ethereum's PoS system, any lapse in ensuring protocol security results in a corresponding slash of their stakes. Furthermore, we consider scenarios where a subset of malicious verifiers might attempt to leak transaction details to Mallory, and demonstrate how FIRST prevents such occurrences in Section 5.

Validators: We assume that the validators are greedy—they sort transactions in descending order of priority fee and pick them in an order that maximizes their profit. It is important to note that in FIRST, verifiers do not have access to client details during $\mathcal{V}$ verification, effectively preventing transaction censorship. They can also re-order transactions to increase their profit and attempt to frontrun victim transactions.

Coordinator: The coordinator is randomly chosen by Alice from a set of verifiers $\mathbb{V}$. It's important to emphasize that while this entity doesn't need to be trusted for security purposes, it is essential for ensuring liveness. We assume Alice actively monitors the transaction process. If any intentional delays are detected, Alice will re-elect a coordinator and continue her interactions with the new entity.

dApp Creator: We assume *dAC* will deploy *SC* and implement it correctly. We also assume that *dAC* does not collude with any other participant or with validators as it is in their best interest to protect their dApp for business reasons. Furthermore, the inherent transparency of smart contract code, which is accessible to the public, acts as a safeguard against malicious intent. Moreover, we assume that *dAC* has both completed the Know-Your-Customer (KYC) process and undergone an audit for the protocol, providing an added layer of deterrence against malicious attempts. While KYC verification is predominantly utilized in centralized services, there are companies like that offer this service for dApps [35]. KYC verification ensures that in the event of any malicious actions by *dAC*, the real-world entity behind it can be easily identified, thereby enhancing the deterrent effect against potential malicious activities.

We do not discuss networking-related attacks as they are out of the scope of this work; we refer the reader to relevant research [29].

4 THE FIRST FRAMEWORK

4.1 Overview of FIRST

The conceptual idea behind FIRST is to prevent Mallory (an attacker) from frontrunning Alice's transaction by ensuring that Mallory's transaction cannot reach the smart contract before Alice's is posted. To achieve this, FIRST requires every user interacting with a FIRST-protected contract to compute a verifiable delay and wait independently for a predetermined time t_1 before submitting their transaction to the mempool. Importantly, FIRST does not impose a global ordering or queue across users; each user's delay is enforced individually. This mechanism guarantees that no transaction becomes visible to adversaries until after the VDF commitment is fulfilled.

The goal is to choose t_1 for a given time period/epoch, s.t. $t_1 >> t_2$, where t_2 is the expected wait time of the transaction of any Alice in the mempool before getting posted on the Blockchain. This ensures that, with high probability, Mallory cannot frontrun Alice's transaction that she sees in the mempool. The time t_2 depends on several dynamic factors, namely transaction gas price, priority fee, miner extractable value (MEV), and network congestion at the time of submission, which makes an exact assessment of t_2 difficult. Since the expected value of t_2 is the best can be done, there is a chance of t_1 being less than the actual waiting time for Alice's transactions. Given that t_2 is difficult to predict, and a high t_1 is detrimental to transaction throughput due to latency, what we do is empirically arrive at a "reasonable" value for t_1. FIRST continuously monitors the blockchain data to identify the minimum priority fee value that would result in a high likelihood of all FIRST transactions waiting approximately t_2 time in the mempool. The t_1 wait time is then fixed for a given epoch (higher than t_2), ensuring that a potential attack transaction has very low probability to frontrun valid FIRST transactions. For our application of FIRST in Ethereum, we set this epoch to be the same as the default Ethereum epoch of 32 blocks. The *dAC* obtains the value of t_1 via statistical analysis of the relation between the priority fee of the transaction and transaction confirmation time by monitoring the Ethereum network continuously. Consequently, FIRST recommends an optimal priority fee that significantly decreases the likelihood of transactions getting frontrun. We detail how we perform such a statistical analysis in Section 6.

4.2 Construction of FIRST

This section outlines the key components of FIRST, illustrated in Figure 2, which consists of seven protocols. **Steps 1–2** correspond to the deployment of the smart contract on the blockchain and the registration of verifiers with the dApp owner. After registration, each verifier independently generates a key pair. These steps constitute the bootstrap phase of the system (Protocols 1 and 2). **Steps 3–4** represent the initialization of a transaction by Alice. She prepares the transaction details and requests a VDF challenge from the verifier set (Protocol 4). **Step 5** shows the verifiers responding with a unique VDF challenge—a prime l—which Alice must use to compute the VDF (Protocol 6). **Steps 6–8** cover the VDF evaluation and verification phase (Protocol 5). Alice computes the VDF output offchain and sends the proof to the verifiers, who verify its correctness and sign the result. **Step 9** corresponds to Alice submitting her transaction, along with the signed proof, to the smart contract on Blockchain (Protocol 7). The contract then verifies both the signature and VDF proof before executing the transaction onchain. For simplicity, the computation of the recommended transaction tip, *FIRST_FEE*, described in Protocol 3, is not visualized in the figure, but it is applied before the final transaction submission.

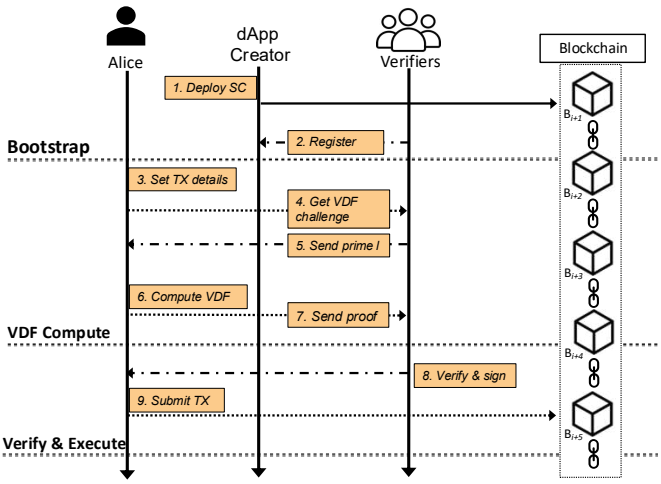


Figure 2: Overview of FIRST.

We use Sign and Verify with no pre-pended string to denote regular digital signature functions, whereas Agg.function() denotes functions specific to the aggregate signature scheme. We use Verify for both, signature and VDF verification, which will be clear from context. Below we discuss each protocol.

Protocol 1: This is the bootstrap protocol of FIRST, and is executed only once. It takes a security parameter as input and outputs the smart contract SC and verification/signing keypairs for each member of $\mathbb{V}$. First, the dAC implements and deploys the dApp on Ethereum. Entities sign up with the dAC to become verifiers. Following deployment, each member of $\mathbb{V}$ generates their key pairs. We note that $\leftarrow$ denotes an assignment operation.

Protocol 2: The protocol is used to generate the system parameters of FIRST. It takes in a security parameter and outputs the public parameters (pp) of VDF $\mathcal{V}$. In Line 1, each $V_i \in \mathbb{V}$ initializes its list D_i and U_i, used to keep track of values used in the VDF $\mathcal{V}$'s evaluation and verification, respectively. In Line 2, the dApp creator dAC initializes the number of steps T that will be used in the evaluation of $\mathcal{V}$. T is the number of steps required to evaluate the VDF instance which results in a corresponding delay of t_1 units of time. Next, dAC samples a negative prime integer d, which satisfies $d \equiv 1 \pmod 4$. These requirements ensure that when generating the class group (Cl) from d in Line 4, the resulting class group order cannot be efficiently computed by any known algorithm [19, 42].

Currently, two approaches are known for setting up $\mathcal{V}$: using an RSA group of unknown order and using class groups of imaginary quadratic fields [42] whose order is hard to determine. The RSA group approach requires a trusted setup when generating N such that $N = p.q$ where p and q are primes; in particular, p and q need to be kept secret. On the other hand, a class group of imaginary quadratic fields does not require a trusted setup and is used by blockchains such as the Chia network in production [19]. We use such groups to eliminate the trusted setup requirement in our construction.

Protocol 1: System setup.

Inputs : Security parameter λ.
Output : SC, (pk, sk) keypair for each member of $\mathbb{V}$.
Parties : dApp creator (dAC), set of verifiers ($\mathbb{V}$).

1 dAC implements SC and deploys it on Ethereum.
2 Each V_i; $i \in [1 \dots n]$ generates $(pk_i, sk_i) \leftarrow \mathsf{Agg.KeyGen}(1^\lambda)$.

Protocol 2: Parameter generation.

Inputs : Security parameter λ.
Output : pp.
Parties : dApp creator (dAC), set of verifiers ($\mathbb{V}$).

1 Each $V_i \in \mathbb{V}$, initializes lists $D_i, U_i = [\ ]$.
2 dAC picks $T \in \mathbb{Z}^+$.
3 $dAC \leftarrow\!\!\$\ d$, s.t., d is negative prime and $d \equiv 1 \pmod 4$.
4 dAC computes $\mathbb{G} \leftarrow \mathrm{Cl}(d)$ and output $pp = (\mathbb{G}, T)$

Protocol 3: The goal of FIRST is to provide users with frontrunning resistant transactions. To achieve this, we introduce a novel mechanism for computing a custom recommended fee, denoted as *FIRST_FEE*. Without *FIRST_FEE*, users would have to rely blindly on wallet or Etherscan-suggested priority fees or manually estimate gas prices, exposing them to potential delay in transaction inclusion, which increases their vulnerability to frontrunning or may lead to unnecessary overpayment. The *FIRST_FEE* helps incentivize faster pickup of the FIRST transactions, thus ensuring their faster confirmation and lower latency. This protocol continuously monitors blockchain transactions to analyze transaction wait times and the associated priority fees paid, in order to calculate the *FIRST_FEE*. The α value passed as input to the protocol corresponds to the weight of simple Exponentially Weighted Moving Average (EWMA) calculation for *FIRST_FEE* inside the CalcFIRSTFee function.

The value of t_1 in Protocol 3 corresponds to the T value set in Line 2 of Protocol 2. T is the estimated number of steps required by a powerful machine (e.g. one with a modern desktop CPU) to compute a VDF proof with t_1 delay. Most other less capable machines will take longer than t_1 to compute the VDF. For each new block posted on the blockchain, Protocol 3 calculates the average fee (f_{avg}) paid by transactions which waited less than t_2 time in the mempool before being posted in the blockchain. The average value calculated in the previous step is then incorporated into the *FIRST_FEE* value using EWMA. The FIRST protocol enables the forceful change of an epoch if the t_2 value needs to be updated before the current epoch ends. For instance, when the number of transactions in the current block waiting less than t_2 time are statistically insignificant or cross a predefined system threshold (in Protocol 3, $temp_{list} == \emptyset$) we initiate epoch change, and update the t_2 and t_1 values based on the average waiting time across a set number of recent past blocks, which can be a system parameter (10 blocks in our experiments).

We note that during the shift from a longer to a shorter delay period (t1), FIRST momentarily halts transaction submissions. This precaution maintains fairness between transactions with varying VDF delays during the transition (t_1 to t_1'), safeguarding transactions with extended VDF delays from being outpaced by those with shorter ones. The pause ensures that all users who started

Protocol 3: FIRST recommended priority fee calc.

Inputs : Initial t_2, α, and k (multiplication factor for t_1).
Output : Recommended priority fee.
Parties : dApp creator (*dAC*).

function CalcFIRSTFee(*FIRST_FEE*,tx_{list})**:**
  $temp_{list}$ = [ ].
  for tx *in* tx_{list} **do**
    if $tx_{wait\_time} < t_2$ **then**
      $temp_{list}.append(tx_{priority\_fee})$.
    end
  end
  if $temp_{list} == \emptyset$ **then**
    /* re-calibrate t_2 & t_1. */
    Initiate epoch change.
    return
  end
  f_{avg} = average($temp_{list}$).
  if *FIRST_FEE == 0* **then**
    FIRST_FEE = f_{avg}.
  end
  else
    FIRST_FEE = $\alpha \times f_{avg} + (1 - \alpha) \times$ *FIRST_FEE*.
  end
function main()**:**
  FIRST_FEE = 0.
  $t_1 = k \times t_2$.
  while *True* **do**
    if *New block with* tx_{list} *transactions is posted on BC* **then**
      CalcFIRSTFee(*FIRST_FEE*, tx_{list}).
      if *Current epoch ended* **then**
        Update t_2 if needed, set $t_1 = k \times t_2$ and update T to correspond t_1.
      end
    end
  end

Protocol 4: Transaction detail generation.

Inputs : $addr_A$, f_{name}, $addr_{SC}$.
Output : Secret message M_A, h, Signature σ_A.
Parties : set of verifiers ($\mathbb{V}$), user in system (Alice).

1 Alice generates message $M_A = (addr_A, f_{\text{name}}, addr_{SC}, input_{SC})$.
2 Alice generates hash of M_A, $h = H(M_A)$, and signs it: $\sigma_A \leftarrow \text{Sign}(sk_A, h)$.
3 Alice sends (h, σ_A) to each V_i; $i \in [1 \dots n]$, $n = |\mathbb{V}|$, including the V_i she picks as the coordinator C for signature aggregation.

Protocol 5: User-Verifiers interaction.

Inputs : σ_A, h, pp.
Output : VDF output y, VDF proof π.
Parties : user in system (Alice), set of verifiers ($\mathbb{V}$) including coordinator C.

1 On receiving (σ_A, h) from Alice, $C \in \mathbb{V}$ picks prime $l \leftarrow\$ \mathbb{P}$ and sends (h, l) to $\mathbb{V} \setminus C$.
2 **for** *each* $V_i \in \mathbb{V}$ **do**
3 **if** $l \notin D_i$ *and* $l \notin U_i$ **then**
4 **if** true $\leftarrow$ Verify(pk_A, h, σ_A) **then**
5 $M_i = (l, h, V_i, block_{curr})$.
6 $\sigma_{V_i} \leftarrow$ Agg.Sign(sk_{V_i}, M_i).
7 Add (l) to D_i.
8 Send (M_i, σ_{V_i}) to C.
9 **end**
10 **end**
11 **end**
/* Sig. aggregation and evaluate $\mathcal{V}$ */
12 C checks if each Agg.Verify$(pk_i, M_i, \sigma_{V_i}) \stackrel{?}{=}$ true. If majority of members of $\mathbb{V}$ return $\perp$, C returns $\perp$ to Alice. Else C does $\sigma_{\text{agg}} \leftarrow$ Agg.Aggregate$(M_1, \dots, M_j, \sigma_{V_1}, \dots, \sigma_{V_j})$, where $j > |\mathbb{V}|/2$.
13 C creates $M_{\text{agg}} = (\sigma_{\text{agg}}, M_1, \dots, M_j, pk_1 \dots pk_j)$ and sends M_{agg} to Alice.
14 Alice checks if Agg.AggregateVerification$(\sigma_{\text{agg}}, M_1, \dots, M_j, pk_1, \dots, pk_j) \stackrel{?}{=}$ true, and if Verify$(pk_i, pp, \sigma_{pp_i}) \stackrel{?}{=}$ true where $i \in \{1 \dots j\}$, and $j > |\mathbb{V}|/2$. If both return yes, Alice computes $(\pi, y) \leftarrow \mathcal{V}$.Eval$(pp, l)$. Else returns $\perp$ and retry.
15 Alice sends (π, y) to all members of $\mathbb{V}$.

their transaction setup under the old t_1 finish their VDF evaluation correctly before the new t_1' becomes active. Without a pause, adversarial users could strategically delay their transaction request to fall after the adoption of t_1' to benefit from the reduced delay parameter, which violates fairness guarantees.

Protocol 4: This protocol is used to generate transaction details of FIRST's users. It takes as input Alice's transaction details and outputs a message, its digest and a signature over the digest; the latter two are meant to be given to $\mathbb{V}$. In Line 1, user Alice constructs a tuple, M_A, with the transaction details, including her Ethereum address $addr_A$, the dApp smart contract address that she intends to submit a transaction to, $addr_{SC}$ (that *dAC* created), and the name of the function that she intends to invoke to trigger the smart contract SC, f_{name}. We assume she has a verification/signing keypair (pk_A, sk_A), using which, in Line 2, she creates and signs a digest of M_A. Using the cryptographic hash of the transaction details prevents the leakage of any detail that may help a potential frontrunner. Alice sends the digest of M_A (h) and her signature over it (σ_A) to each V_i; $i \in [1 \dots n]$. Alice chooses coordinator (C) from $\mathbb{V}$ to help with signature aggregation in Protocol 5 and Protocols 6.

Protocol 5: This protocol must be executed between Alice and members of $\mathbb{V}$. It takes as input the output of Protocol 4, i.e., the digest/signature over Alice's message. It outputs the evaluation of the VDF instance, $\mathcal{V}$, and its corresponding proof. In Line 1, the coordinator C samples a unique (per user) prime l from a set of primes $\mathbb{P}$ that contains the first $2^{2\lambda}$ primes. We require each V_i to

Protocol 6: VDF verification and tx submission.

Inputs : π, y.
Output : Aggregate signature σ'_{agg}, transaction tx_A.
Parties : user in system (Alice), set of verifiers ($\mathbb{V}$).

/* Each verifier runs proof verification */
1 **for** *each* $V_i \in \mathbb{V}$ **do**
2 **if** $l \notin U_i$ *and* $l \in D_i$ **then**
3 Add l to U_i.
4 **if** *"accept"* $\leftarrow \mathcal{V}$.Verify*(pp, l,* y, π*)* **then**
5 $M'_i = (\text{"}accept\text{"}, V_i, l)$.
6 $\sigma'_{V_i} \leftarrow \text{Agg.Sign}(sk_{V_i}, M'_i)$.
7 Send (M'_i, σ'_{V_i}) to C.
8 **end**
9 **end**
10 **end**
/* Sig. aggregation and submit tx. */
11 C checks if each $\text{Agg.Verify}(pk_i, M'_i, \sigma'_{V_i}) \stackrel{?}{=} \text{true}$. If majority of members of $\mathbb{V}$ return $\perp$, C returns $\perp$ to Alice. Else C does $\sigma'_{\text{agg}} \leftarrow \text{Agg.Aggregate}(M'_1, \ldots, M'_j, \sigma'_{V_1}, \ldots, \sigma'_{V_j})$, where $j > |\mathbb{V}|/2$.
12 C creates $M'_{\text{agg}} = (\sigma'_{\text{agg}}, M'_1, \ldots, M'_j, pk_1, \ldots, pk_j)$ and sends to Alice.
13 Alice checks if $\text{Agg.AggregateVerification}(\sigma'_{\text{agg}}, M'_1, \ldots, M'_j, pk_1, \ldots, pk_j) \stackrel{?}{=} \text{true}$ and $j > |\mathbb{V}|/2$, if yes, Alice creates $M' = (M_A, M_{\text{agg}}, M'_{\text{agg}})$ and signs it, $\sigma'_A \leftarrow \text{Sign}(sk_A, M')$. Else returns $\perp$ and retry.
14 Alice retrieves the current recommended priority fee (*FIRST_FEE*) from Protocol 3.
15 Alice creates and submits transaction $tx_A = (\sigma'_A, M', pk_A, \textit{FIRST\_FEE})$.

Protocol 7: Signature validation and SC execution.

Inputs : tx_A, $block_{now}$ and $threshold$.
Output : Smart Contract Functionality.
Parties : User in system (Alice), Smart Contract (*SC*).

1 Parse $tx_A = (\sigma'_A, M', pk_A, \textit{FIRST\_FEE})$,
$M' = (M_A, M_{\text{agg}}, M'_{\text{agg}})$,
$M_{\text{agg}} = (\sigma_{\text{agg}}, M_1, \ldots, M_j, pk_1, \ldots, pk_j)$, and
$M'_{\text{agg}} = (\sigma'_{\text{agg}}, M'_1, \ldots, M'_j, pk_1, \ldots, pk_j)$, where $j > |\mathbb{V}|/2$.
2 **if** $(H(addr_A, f_{name}, addr_{SC}, input_{SC}) \stackrel{?}{=} h)$ *and* $((l, h, \cdot, block_{curr}) \in [M_1, \ldots, M_j])$ *and* $((\text{"}accept\text{"}, \cdot, l) \in [M'_1, \ldots, M'_j])$ *and* $(|M_1, \ldots, M_j| > |\mathbb{V}|/2)$ *and* $(|M'_1, \ldots, M'_j| > |\mathbb{V}|/2)$ **then**
3 **if** $\text{Agg.AggregateVerification}(\sigma'_{\text{agg}}, M'_1, \ldots, M'_j, pk_1, \ldots, pk_j) \stackrel{?}{=} \text{true}$ **then**
4 **if** $block_{now} - block_{curr} < threshold$ **then**
5 *SC* executes the intended functionality.
6 **end**
7 **end**
8 **end**

independently check l and verify that it was not generated before (Lines 2, 3).

Upon checking the validity of l and Alice's signature, each V_i creates a message M_i by concatenating l, $block_{curr}$, and h from Protocol 4 to its id and signs M_i (Lines 5, 6). $\mathcal{V}$'s freshness $block_{curr}$, which represents the block height at the time of request is included in M_i to prevent off-line attacks on $\mathcal{V}$. For an off-line attack, Mallory requests l and pre-evaluates the $\mathcal{V}$ to submit the frontrunning transaction when the victim transaction is seen on the network. However, the smart contract eliminates this attack by verifying the freshness of $\mathcal{V}$. In Line 7 and 8, each $V_i \in \mathbb{V}$ updates their D_i to keep track of used l values and sends their σ_i to C. The list D_i is used to ensure that no user in the system has been given the current l for $\mathcal{V}$ computation, else colluding users can reuse proofs. The list U_i is used to ensure that users in the system can only use a given l once, hence thwarting any replay attacks. In Lines 18 and 19, C verifies the signatures of verifiers, aggregates them, and sends the aggregate signature to Alice for verification. We note that both D and U are public lists.

The goal of the aggregate signature scheme in FIRST is to cut down the cost of verifying each V_i's signature individually. Moreover, we can obtain aggregate signatures from all members of $\mathbb{V}$ without requiring any trust assumption on them. We refer interested readers to [15] for further details on the aggregate signature scheme. Alice checks the validity of σ_{agg} and the number of received messages j, where $j > |\mathbb{V}|/2$. If both return true, Alice retrieves and verifies the public parameters of $\mathcal{V}$, pp, and starts the evaluation of $\mathcal{V}$ (we recollect that per our system model, Alice evaluates $\mathcal{V}$). During the evaluation, Alice generates output and proof of correctness π, which is sent to all members of $\mathbb{V}$ (Lines 20, 21).

Protocol 6: Protocol 6 is required to be executed between Alice and $\mathbb{V}$. It takes as input the VDF evaluation result and its proof (given as output by Protocol 6), the pp of $\mathcal{V}$ and outputs Alice's transaction tx_A to be submitted to *SC*. In Line 2, every $V_i \in \mathbb{V}$ first checks if $l \notin U_i$. This check ensures that Mallory is not reusing the l to evaluate $\mathcal{V}$. Each V_i also checks if $l \in D_i$, to check if l has indeed been assigned to a user. If the check returns true, every $V_i \in \mathbb{V}$ adds l to U_i. In Line 4, every $V_i \in \mathbb{V}$ verifies the VDF proof π sent by Alice. Depending on the outcome of the verification, each V_i creates M'_i and signs it (Lines 5, 6). Upon completion of the verification phase, in Line 17, C first verifies each σ'_i and aggregates the signatures into a unique signature σ'_{agg}. In Line 18, C creates a tuple M'_{agg}, containing the σ'_{agg}, distinct messages of members of $\mathbb{V}$, their public keys, and sends it to Alice. Alice checks the validity of σ'_{agg} and the number of received messages j, where $j > |\mathbb{V}|/2$, for a majority of verifiers from $\mathbb{V}$. [1] If both return true, Alice creates M' consisting of her message, M_A from Protocol 4, M_{agg} from Protocol 6, and M'_{agg} from Protocol 5. Alice signs it before creating transaction tx_A, sets the transaction fees (*FIRST_FEE* from Protocol 3 and current Ethereum base fee), and submits the transaction (Lines 19, 20, 21).

Protocol 7:This protocol is used to validate the transaction tx_A, $\mathcal{V}$'s verification details, and the signature aggregation. Alice creates

[1]The number of messages received by Alice in Protocol 6 and Protocol 5 are both denoted by j, but we note that the value of j in both protocols need not be exactly the same, as long as it satisfies the property $j > |\mathbb{V}|/2$.

and submits tx_A with recommended fee. *SC* parses tx_A to access necessary fields. *SC* verifies transaction details committed to in Protocol 4, verifies the messages of verifiers and checks if the number of participants in the verification phase is more than $|\mathbb{V}|/2$. Finally, *SC* will check the given $\mathcal{V}$'s freshness by checking if the difference between the block height at the time of request and the current lies within a pre-defined system threshold that should be adjusted by *dAC*. We note that SC examines all messages and employs the $block_{curr}$ value endorsed by the majority to validate the freshness of VDF, rather than relying on single $block_{curr}$. The *SC* will abort the function execution if any check fails.

5 SECURITY ANALYSIS OF FIRST

5.1 Informal Security Analysis

In this section, we analyze the security of FIRST informally by considering potential attack scenarios and describe how FIRST eliminates them.

Malicious Verifier: In this attack, an adversary might try to corrupt some members of $\mathbb{V}$, and try to glean information about Alice's transaction tx_A while she computes the VDF. FIRST accounts for this by having Alice conceal all transaction details by hashing them and sharing only the digest with the verifiers (Protocol 4, Steps 2-3), thus preventing any leakage of sensitive information. The general security guarantees apply for the case where a malicious verifier attempts to frontrun Alice.

Proactive Attacker: Consider a scenario where Mallory or a bot she created is monitoring the pending transaction pool to identify a transaction tx_A submitted by a user Alice. Let t_M represent the time Mallory first sees tx_A, with a gas price G_A, on the pending pool. Mallory creates a transaction tx_M with gas price G_M where $G_M > G_A$. We note that, in order for this attack to succeed, tx_M is required to be included in the previous or in the same block but before tx_A. To address this, FIRST assigns $\mathcal{V}$ related parameters and updates them regularly using the empirical analysis we describe in Section 6. Since all valid transactions need to wait for FIRST stipulated time delay ($\mathcal{V}$ delay), tx_M will need to wait to generate valid $\mathcal{V}$ proof. If Alice paid the FIRST recommended priority fee, tx_A will wait for at most t_2 time in the pending pool, and since t_2 is less than the $\mathcal{V}$ delay set by FIRST (t_1), Alice's transaction will not get frontrun by Mallory with high probability.

Backrunning and Sandwich attack: Backrunning is another attack strategy where Mallory creates a transaction tx_M with a gas price of G_M where $G_M < G_A$ to take advantage of the outcome of Alice's transaction [32]. Given the enforced $\mathcal{V}$ delay, the malicious transaction attempting to backrun the victim transaction has to wait before entering the mempool, which prevents backrunning, making it impossible for the attacker's transaction to be scheduled in the same block thus preventing frontrunning. Given both frontrunning and backrunning are prevented, sandwich attack is also prevented [44].

Malicious Block Proposer: In Ethereum 2.0, the block proposers are randomly chosen from active validators whose aim is to propose a potential block for the slot they are assigned. As a result, they have full control over inserting, excluding, and re-ordering transactions akin to miners in the PoW version of Ethereum. A potential attack can be frontrunning transaction inserted into the block by the malicious block proposer. However, FIRST already handles this case: if any transaction does not contain the aggregated signature of $\mathbb{V}$ on the verification of $\mathcal{V}$ proof, the smart contract will reject the transaction.

Impact on Blockchain Throughput: In EVM-based blockchains, throughput is constrained by the block gas limit—30 million gas on Ethereum—with new blocks produced roughly every 12 seconds. Since VDF computation and verifier coordination are performed entirely off-chain, only the final, verifier-signed transaction is submitted on-chain. As a result, FIRST does not affect blockchain throughput. Once submitted, a FIRST transaction behaves like any other following the standard inclusion and confirmation processes.

Pre-computed VDF attack: Attackers may attempt to create frontrunning transactions in advance and broadcast them when they see the victim transaction in the pending pool. We eliminate this pre-computation attack vector by checking the freshness of the VDF during smart contract execution (Line 4, Protocol 7). Specifically, suppose the difference between the current block (where the transaction is slated for execution) and the block height at the time of the transaction request is greater than a pre-defined system threshold, the transaction will be reverted. Verifiers may optionally charge a small fee for issuing a new prime l for the VDF to create an economic disincentive against repeated VDF challenge requests. Since verifier interaction occurs offchain, standard Web2 techniques—such as IP-based rate limiting or user-level quotas—can also be applied to mitigate abuse.

5.2 Formal Security Analysis

We analyze the security of FIRST in the Universal Composability (UC) framework [17]. To this end, we define an ideal functionality, $\mathcal{F}_{\text{FIRST}}$, consisting of three functionalities, $\mathcal{F}_{\text{setup}}$, $\mathcal{F}_{\text{bc}}$, and $\mathcal{F}_{\text{construct}}$ along with two helper functionalities $\mathcal{F}_{\text{sig}}$ [17] and $\mathcal{F}_{\text{vdf}}$ [27]. We assume that the optimal functionalities share an internal state and can access each other's stored data. We prove the following theorem in the Appendix C.

Theorem 5.1. *Let $\mathcal{F}_{FIRST}$ be an ideal functionality for FIRST. Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary for FIRST, and let $\mathcal{S}$ be an ideal-world PPT simulator for $\mathcal{F}_{FIRST}$. FIRST UC-realizes $\mathcal{F}_{FIRST}$ for any PPT distinguishing environment $\mathcal{Z}$.*

6 EXPERIMENTAL RESULTS AND ANALYSES

We evaluate the performance of FIRST on real Ethereum traces over a month long period of observation. We analyze FIRST's suggested *FIRST_FEE* during our experiment and show the effectiveness of FIRST in terms of the percentage of frontrunnable transactions in a given time period.

A low percentage implies that transactions submitted during the said time period with *FIRST_FEE* are seldom frontrun. The success of FIRST is not only dependent on the FIRST system parameters, namely k, $alpha$, and t_2, but also on the system-specific network dynamics. We replicate our analysis of FIRST over a non-EIP-1559 chain, Binance Smart Chain (BSC). In what follows, we discuss details of our experimental setup, data gathering, and experimental results.

Functionality $\mathcal{F}_{\text{bc}}$

Miner p_i requesting current d_i: Upon receiving $(RequestRound, sid)$ from p_i, send d_i to p_i.
Adversary corrupting Miner p_i: Upon receiving $(corrupt, p_i, sid)$ from $\mathcal{A}$, if $|\ \mathbb{H} \setminus \{p_i\}\ | > \mathbb{P}/2$ then set $\mathbb{H} := \mathbb{H} \setminus \{p_i\}$, else return $\perp$.
Block hashing: When ctrTime == bcHashTime, $\mathcal{F}_{\text{bc}}$ takes a set of tuples $\mathbb{TX}_B$ such that $\mathbb{TX}_B \subseteq \mathbb{TX}_P$ where $\mathbb{TX}_P$ represents the set of transactions in txpoolTable, and $|\mathbb{TX}_B| = l$. $\mathbb{TX}_B$ is picked such that $l = min(|\mathbb{X}|, \forall \mathbb{X} \in \mathcal{P}(\mathbb{TX}_P))$ and $\sum_{i=1}^{l} tx_i.txfee \lessapprox$ blockMaxFee where $\mathcal{P}$ represents a power set function. $\mathcal{F}_{\text{bc}}$ then adds blockNum to each tuple (e.g. tuple $(tx,$ blockNum$)$ $\forall\ tx \in \mathbb{TX}_B$) and moves them to bcTable and sends to $\mathcal{S}$ and $\mathcal{A}$. $\mathcal{F}_{\text{bc}}$ sets ctrTime = 0 and blockNum = blockNum + 1.
BC data request handling: $\mathcal{F}_{\text{bc}}$ on receiving request $(\text{getData}, sid)$ from user u, retrieves all data tuples from bcTable, txpoolTable, and scTable, and sends to u, $\mathcal{S}$.
BC block num request handling: $\mathcal{F}_{\text{bc}}$ on receiving request $(\text{getBlockNum}, sid)$ from a user return blockNum.
Initialization of *BC***:** On receiving $(init, sid, \mathbb{P}, \mathbb{H},$ blockMaxFee, bcHashTime$)$ from $\mathcal{Z}$, initialize for each BC miner/validator $p_i \in \mathbb{P}$ a bit $d_i := 0$, sets blockMaxFee as the max fee limit for each block, sets current block hashing interval time as bcHashTime, sets ctrTime = 0, and set blockNum = 0. Set $\mathbb{H} \subset \mathbb{P}$ to be set of honest validators.
Smart contract deployment: $\mathcal{F}_{\text{bc}}$ on receiving $(sid, \text{deploy}, SC.id, code)$ from any node stores the tuple $(SC.id, code)$ in an scTable for later retrieval and execution. The *code* of $SC.id$ will eventually call $\mathcal{F}_{\text{setup}}$ to verify the hash of M_u in hashTable and the aggregate signature in sTable of some submitted transaction $tx = (M_u, \sigma_u, SC.id, (\sigma_{\text{agg}}, M_1, \ldots M_n, pk_1, \ldots, pk_n), txfee)$ and check that majority of $M_1, \ldots, M_n$ contain an $(\text{“}accept\text{”}, \cdot, \cdot)$. If verification fails, then SC outputs a failure tx', else it continues execution of the $SC.id$ *code* which will include verifying hash h of the submitted transaction M_u, and finally outputs a successful tx'.
Transaction request handling: $\mathcal{F}_{\text{bc}}$ on receiving (sid, invoke, tx) stores the tx tuple in txpoolTable. If the tx is invoking a smart contract $SC.id$, then $\mathcal{F}_{\text{bc}}$ retrieves the tuple $(SC.id, code)$ from scTable. $\mathcal{F}_{\text{bc}}$ executes *code* with the given tx and the output transaction tx' is generated. Both transactions are added to txpoolTable and also sent back to user u and $\mathcal{S}$. All rows in txpoolTable are arranged in descending order of $txfee$ at all times.
Miners stepping the time counter forward: Upon receiving message $(RoundOK, sid)$ from party p_i set $d_i := 1$. If for all $p_j \in \mathbb{H}$: $d_j = 1$, then reset $d_j := 0$ for all $p_j \in \mathbb{P}$ and set ctrTime = ctrTime + 1. In any case, send $(switch, p_i)$ to $\mathcal{A}$. The adversary is notified in each such call to allow attacks at any point in time.

Figure 3: Ideal functionality for blockchain.

6.1 Data Gathering

In order to get the most accurate waiting times of transactions in the pending pool, we deployed a Geth[2] full node (v.1.11.0) running on an Amazon AWS Virtual Machine located in North Virginia. The AWS node had an AMD EPYC 7R32 CPU clocked at 3.30 GHz with 8 dedicated cores, 32 GB of RAM, 1.3 TB solid-state drive, running Ubuntu (v.20.4). We also ran a beacon node using Prysm[3] (v.3.1.2) software which is required to coordinate the Ethereum proof-of-stake consensus layer operations. Once the deployed node synced, we collected the data in the Geth node's pending pool. The data collected included transaction arrival times and the transactions' corresponding unique transaction hashes from block number 15665200 to block number 15886660[4]. For each collected transaction from the confirmed blocks on the blockchain, we gathered additional details such as block base fee, paid max priority fee, gas price, and block confirmation time. Although the data is from October 2022, the transaction volume has remained stable since then (see [9]), and Ethereum still lacks frontrunning protection at the application layer, making the data relevant.

We used a machine with Apple M1 Max chip, 32 GB RAM, 1 TB HDD, running macOS Monterey (v.12.6) to perform experiments on the collected data. There were a total of 30.6M transactions for the given block range (15665200–15886660), out of which, 24.34M were Type-2 (EIP-1559) transactions and 6.26M were Type-1 (legacy, non-EIP-1559) transactions. We analyzed the more common Type-2 transactions, FIRST can also be used for Type-1 transactions. Out of the total 30.6M transactions our node was able to detect the wait time for 29.65M transactions. Since our node did not receive a total of 944807 transactions (roughly 3.08%), we conclude that these transactions were either never sent to the P2P layer because of the use of relayers (e.g., Flashbots) or our node did not receive them before their confirmation on the blocks due to network latency. In practice, the dApp owner would deploy multiple full nodes to collect the pending pool data, hence minimizing the chance of missing transactions due to network latency. We deployed another full Geth node in AWS in Singapore with the same software and hardware specifications as the one in North Virginia. The intent was to perform a comparative sanity-check on the transactions copies recorded at two geographically diverse locations.

We computed the waiting times of transactions received by our node by subtracting the transaction's block confirmation time from the recorded time when the transaction was first seen in our node's pending pool. The difference in waiting times of transactions in the US and Singapore was very small. Across all the transactions that we captured, the difference between the receipt times in the US and Singapore was no more than 2 ns for any transactions. Interestingly, our Singapore node also never received any of the 944807 transactions that were not seen by our US node, leading us to conclude that those transactions were privately relayed.

6.2 Extension to non-EIP-1559 chain

Many Ethereum Virtual Machine (EVM) based blockchains, such as Polygon and Fantom, have implemented the EIP-1559 patch. Despite

[2]https://github.com/ethereum/go-ethereum
[3]https://github.com/prysmaticlabs/prysm
[4]https://web3js.readthedocs.io/en/v1.2.11/web3-eth-subscribe.html

Functionality $\mathcal{F}_{\text{construct}}$

User request: Upon receipt of tuple (sid, req, h, σ_u) from a user u with identifier uid, $\mathcal{F}_{\text{construct}}$ adds (uid, h, σ_u) to uTable, and returns "success" to u, and forwards $(sid, \text{req}, uid, h, \sigma_u)$ to $\mathbb{V}$ and $\mathcal{S}$.

User response: Upon receiving $(sid, \text{aggregated}, \sigma_{\text{agg}}, uid)$ from C, $\mathcal{F}_{\text{construct}}$ looks for a tuple $(\sigma_{\text{agg}}, \cdot, \cdot)$ in sTable; if such a tuple exists $\mathcal{F}_{\text{construct}}$ retrieves tuple $(l, uid, \cdot)$ from cTable, constructs and returns tuple $(sid, l, \sigma_{\text{agg}}, \cdot, \cdot)$ to uid and to $\mathcal{S}$, else returns $\perp$ to both.

User verification: Upon receiving $(sid, \text{verify}, l, p, s)$ from uid, $\mathcal{F}_{\text{verify}}$ checks if $(l, uid, \text{"not-used"})$ exists in cTable; if yes, updates tuple in cTable to $(l, uid, \text{"used"})$ and forwards (sid, l, p, s) to each $V_i \in \mathbb{V}$ and $\mathcal{S}$, else $\mathcal{F}_{\text{verify}}$ returns $\perp$ to Alice and $\mathcal{S}$.

Coordinator request: Upon receipt of a message (sid, l, uid) from C, $\mathcal{F}_{\text{construct}}$ checks if there exists a tuple (uid, h, σ_u) in uTable. If not, return $\perp$ to C and $\mathcal{S}$. If $(l, \cdot, \cdot)$ exists in cTable, return (sid, fail, l) to C and $\mathcal{S}$, if $(l, \cdot, \text{"used"})$ already exists in cTable, return (sid, used, l) to C and $\mathcal{S}$. Else, $\mathcal{F}_{\text{construct}}$ adds $(l, uid, \text{"not-used"})$ to cTable, $\mathcal{F}_{\text{construct}}$ retrieves (u, pk_u) from idTable, constructs tuple $(sid, \text{valid}, l, h, \sigma_u, pk_u)$ and forwards to all $V_i \in \mathbb{V}$ and $\mathcal{S}$.

Coordinator response: Upon receiving $(sid, V_i, m_i, \sigma_{V_i})$ from members of $\mathbb{V}$, $\mathcal{F}_{\text{construct}}$ forwards $(sid, V_i, m_i, \sigma_{V_i})$ to C and $\mathcal{S}$.

Figure 4: Ideal functionality for transaction processing and VDF construction.

the overall trend of EVM-based blockchains adopting EIP-1559, for completeness, we also studied a non-EIP-1559 chain protocol. We replicated our analysis on the Binance Smart Chain (BSC) which is currently a non-EIP-1559 chain. We deployed a Geth node (v.1.1.17) on AWS Singapore and recorded transaction wait times for 45K blocks (23285229–23288229), totaling 5.29M transactions (statistically significant). Out of the 5.29M transactions, our node did not receive 141157 (2.66%). In non-EIP-1559 chains, the gas price is used to incentivize the validator to pick up a transaction. Hence, FIRST uses gas price to calculate the *FIRST_FEE* in Protocol 3.

6.3 Aggregate Signature Implementation

To assess the cost associated with verifying FIRST transactions via a smart contract, we implemented the aggregated signature verification function [15] using the Solidity programming language and deployed it within a smart contract.

We used elliptic curve pairing operations, such as addition, multiplication, and pairing checks introduced by Ethereum in the form of precompiled contracts with EIP-197 [5]. In Ethereum, precompiled contracts enable the deployment of computationally-intensive operations at a lower cost compared to the users implementing them on their smart contracts. Our implementation uses the `alt_bn128` curve. We used the bn256[6] library (v.0) and the Go programming language (v.1.17.5) to implement the aggregate signature generation and verification schemes. Table 1 shows our results for the verification of the aggregated signatures by the smart contract with different numbers of verifiers. For example, it costs 621180 units of gas for ten verifiers to verify the aggregated signature. Using the median gas cost of **10 GWei** (representative of current market conditions as of December 2024) and the current Ether price of **$3300**, the cost to verify the aggregated signature of 7 verifiers is approximately **$15.65**.

Table 1: GAS CONSUMPTION FOR AGGREGATE SIGNATURE VERIFICATION ON SMART CONTRACT.

Number of Verifiers	Total Gas Consumption (gas units)
5	374423
7	474327
10	621180
15	871987
20	1122943

A 2021 study found that frontrunning extracted approximately $18 million across 200,000 transactions, or about $90 per transaction [37]. FIRST adds an overhead of $10–$15 per transaction, mainly due to the on-chain signature verification. This cost represents a 15% premium per protected transaction. The primary contributor to the cost is the pairing operations required for signature verification. Currently, **EIP-197** is the only supported precompiled contract for pairing operations on Ethereum, using the `alt_bn128` curve. This limited support contributes to the high gas costs. While proposals like **EIP-2537** aim to introduce more efficient cryptographic primitives such as `BLS12-381`, they have not yet been implemented on Ethereum mainnet as of May 2025. Once adopted, these upgrades are expected to significantly reduce verification costs. In addition, designing efficient aggregate-signature schemes is an area of continuing research—we anticipate that the schemes to become more efficient in the future.

FIRST serves as a proof-of-concept demonstrating frontrunning protection on EVM-compatible chains. Additional savings are possible using *gas golfing*[7]. We will assess that in the future. A note of caution is that inline assembly is error-prone and a known source of vulnerabilities in smart contracts [18].

6.4 Scalability of VDF

We assess the practicality of VDF on devices with varying computational capabilities by comparing the VDF computation times on these devices. The VDF [42] used in FIRST has a complexity of $O(T)$ for VDF proof generation and $O(\log T)$ for verification, where T is the number of steps required for proof generation.

For client-side costs experiments, we use a rack server, specifically the PowerEdge R650 Intel Xeon Gold 6354 with 18 cores and 36 threads per core, equipped with 256 GB RDIMM and NVIDIA Ampere A2. Additionally, we evaluated the performance on an iPhone 12 with an A14 Bionic 6-core CPU, 64GB storage, and 4GB RAM, as well as the MacBook Pro. Detailed specifications for the MacBook Pro used in our evaluation can be found in Section 6.1. For this experiment, we choose the sequential steps amount (T) to

[5] https://eips.ethereum.org/EIPS/eip-197

[6] https://pkg.go.dev/github.com/cloudflare/bn256

[7] Gas golfing refers to low-level optimizations such as inline assembly.

be 1 million whereas the bit length of security parameters to be 2048 bits. We give our results in Figure 5. While Figure 5 shows that rack servers compute VDFs faster than other devices, the difference is not sufficient to enable successful frontrunning. This is because no entity, regardless of computational resources, can see the transaction details until after the user's VDF computation is complete and the transaction enters the pending pool. Even highly resourceful devices do not have enough time to complete a new VDF computation fast enough to frontrun, especially when the user includes the optimized priority fee recommended by *FIRST_FEE*.

6.5 Analyses and Discussion

We plot Figure 6a and Figure 6b to demonstrate how FIRST recommended fee changed over our observation period in Ethereum and BSC blockchains, respectively. The figures show the recommended fee (*FIRST_FEE*) on the Y-axes for the corresponding block number on the X-axes, computed using Protocol 3. In both experiments, $k = 3$ and $\alpha = 0.6$. For Ethereum t_2 was 30s and for BSC it was 5s. The *FIRST_FEE* calculated on Ethereum refers to the recommended priority fee, while on BSC, it refers to the recommended gas price. In Figure 6a, the X-axis represents the 198K blocks on the Ethereum blockchain. As seen from the graph, the highest spike in our recommended fee is around block number 15697567. Some blocks have an associated spike in the recommended transaction *FIRST_FEE* due to the surge in the priority fees paid by transactions in the prior blocks. For example, the sale of tokens for the popular NFT project Art Blocks was confirmed in block number 15697567. Out of the 446 transactions in this block, 405 purchased tokens using the Art Blocks contract and paid much higher priority fee than other network transactions. This affected the *FIRST_FEE* for 15697568. Similarly, the second-highest spike around block number 15741444 was due to the NFT project "BeVEE - Summer Collection" sales.

The X-axis in Figure 6b represents the 41K blocks on the BSC blockchain. We see a spike in the FIRST fee for block 23298282 because four transactions indexed in the first four spots of the block 23298281 paid an average of 858.34 GWei in gas fee—escalating the recommended FIRST fee. On analyzing the block, we believe that the transactions paid high fee to profit from arbitrage opportunity.

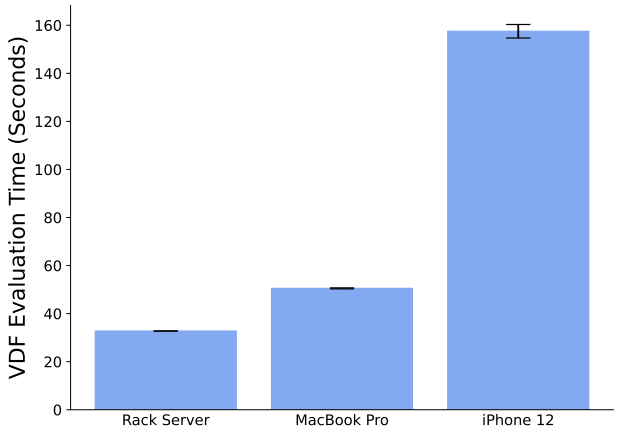


Figure 5: Comparison of VDF computation times across multiple devices.

Despite the unpredictable events in the Blockchain, Figures 6a and 6b show that the computed *FIRST_FEE* adjusts to network activities. In general, we noticed significantly less number of spikes in BSC, compared to Ethereum. This is due to the fast confirmation of transactions in BSC–more discussion at the end of this section.

To initiate our experiments, we obtained the 50^{th} percentile of the maximum wait time for the first 100 blocks and to better handle system dynamism, set t_2 to twice the value, $t_2 = 30$secs. We also analyzed the Ethereum data for $\alpha = \{0.1, 0.2, 0.4, 0.6, 0.8\}$ and found that the $\alpha = 0.6$ gives us better success rate than other values. Note that despite the occasional spikes most transactions pay a low priority fee, hence the value of α has limited impact.

For our analysis, we set $k = 3$, resulting in the VDF delay $t_1 = 90$secs. To reiterate our use cases discussion (Section 7), the VDF delay value is a function of the application and its risk appetite and can be tuned in FIRST. Even with $t_1 = 30$secs, only 0.004% of transactions were susceptible to frontrunning! We discuss this below. Let tx_i represent the i^{th} transaction in a block (b), where $tx_i.fee$ and $tx_i.ctime$ are the transaction fee and the duration tx_i waited on the mempool respectively, and T_b represents the number of transactions in block b. Then, the fraction of potentially frontrunnable transactions in b is given by,

$$fr = \frac{\sum_{i=0}^{T_b} [\![tx_i.fee \geq FIRST\_FEE]\!] [\![tx_i.ctime \geq t_1]\!]}{T_b}, \quad (1)$$

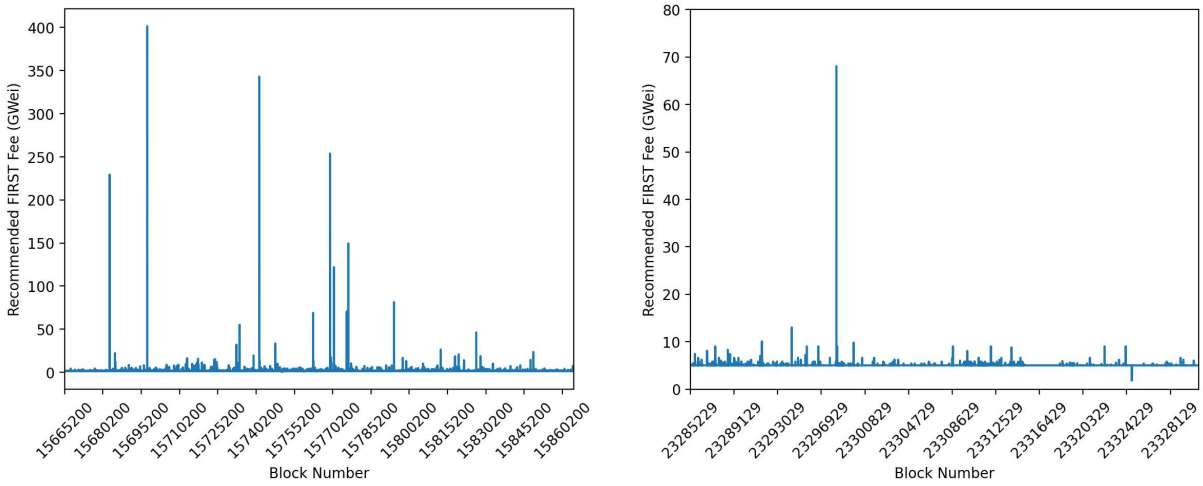
where $[\![.]\!]$ indicates *Iverson brackets* such that $[\![i_{fee} \geq tip_e]\!]$ is *true* (1) if $i_{fee} \geq tip_e$, is *false* (0), otherwise.

We analyzed the Ethereum and BSC data for different values of k. Figure 7 shows the percentage ($fr \times 100$) of transactions that are frontrunnable out of the total transactions (24.34M in Ethereum and 5.14M in BSC) for different values of k. With the VDF delay of 90s ($k = 3$) and the FIRST recommended fee, on the Ethereum blockchain, 196319 out of 198235 blocks (> 99%) had no frontrunnable transactions! With $t_1 = 15$secs ($k = 3$) and the FIRST recommended fee per BSC block, in BSC none of the transactions were frontrunnable. In fact, the percentage of frontrunnable transactions goes to zero for $k \geq 2$. Our choice of $k = 3$ for the data is a good balance between the success rate and the imposed transactions delay.

As we discussed before, on Ethereum, the chance of transactions being frontrun is a bit higher on account of higher volatility (we theorize, due to NFT transactions and slower block confirmation time) compared to BSC, which is more stable on account of the faster settling of transactions. For example, from our data, in the time it takes Ethereum to confirm one block, BSC confirms on an average 4.4 blocks. Each Ethereum block in our dataset has on an average 151 transactions, whereas it is 120 transactions in each BSC block. Thus, 666 BSC transactions are confirmed in the same time as 151 Ethereum transactions.

7 DESIGN CHOICES, COMPATIBILITY, USE CASES AND LIMITATIONS

In this section, we explore potential alternative solutions and their disadvantages, the design choices of FIRST, its compatibility with other protocols, and the limitations of our work.



(a) Recommended FIRST Fee for Ethereum.

(b) Recommended FIRST Fee for BSC.

Figure 6: Recommended FIRST fee for Ethereum and BSC blockchains per block.

Verifiers: One might question the need for a VDF in the presence of an honest-majority committee, which can be used to verify delays. A verifier-based approach would require each verifier to maintain a timer per request, which becomes infeasible as the transaction volume scales. Moreover, such delays cannot be independently verified. In contrast, VDFs are publicly verifiable and efficient to check.

Compatibility with private transaction: The FIRST framework is designed to protect the transaction from getting frontrun. Since it does not change transaction structure, it is compatible with private relayers, such as Flashbots [30]. The only requirement for a transaction before its submission to the relayers is to include the aggregated signature of $\mathbb{V}$ on the verification of $\mathcal{V}$ proof (Protocol 5, Line 20). The *SC* will assert if the transaction includes the aggregated signature and rejects it if not present. FIRST independently prevents frontrunning attacks on EVM-based blockchains without needing extra protocols. While compatible with Flashbots, combining them is redundant and could compromise security through relayer delays.

Potential use cases: Ethereum Name Service (ENS) [4] aims to map long and hard-to-memorize Ethereum addresses to human-readable identifiers. Recent sale trends and exorbitant offers, such as *amazon.eth*, which received a million-dollar offer [5], indicate the importance of frontrunning prevention solutions. FIRST can be used during the sale of these domain names to prevent frontrunning. Non-fungible tokens (NFTs) are unique cryptographic tokens that live on blockchains and are not possible to forge. One of the largest NFT marketplace Opensea exceeded 10 billion dollars in NFT sales in the third quarter of 2021 [6]. Not surprisingly, frontrunning bots are watching the mempool for NFT sales to create a counter transaction to frontrun. One can employ FIRST to prevent such attacks on the marketplaces.

Compatibility with Time-Sensitive Applications: FIRST can be integrated with time-sensitive dApps, such as DEXes and NFT marketplaces, to prevent frontrunning attacks. When a platform adopts FIRST, all user transactions experience a uniform VDF delay, ensuring that transactions are properly time-shifted and fairly ordered. The VDF delay can be tuned to balance responsiveness and security, minimizing user inconvenience while maintaining strong protection guarantees. In the event of a price discrepancy between a FIRST-protected DEX and other exchanges, arbitrageurs will naturally intervene to close the price gap, ensuring consistency across platforms without undermining the protocol's security.

Limitations: Adjusting the real-world delay time with the given VDF delay parameter for every user's computational capabilities is a challenging and open-research problem [12]. While it is an orthogonal task to ours, FIRST mitigates the problem by picking the $t_1 >> t_2$ — this ensures that a more-capable Mallory cannot frontrun a less-capable Alice. The VDF delay parameter is selected

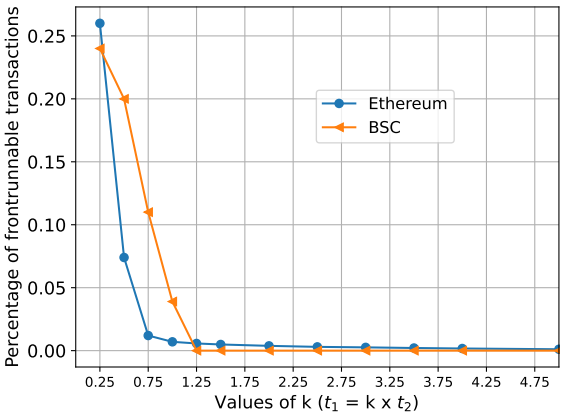


Figure 7: Percentage of frontrunnable transactions (Y-axis) for different values of FIRST parameter k ($t_1 = k \times t_2$).

based on observed transaction settlement times onchain and the computational capabilities of high-end contemporary machines. While not bulletproof, our analysis shows that over 99% of transactions are no longer susceptible to frontrunning under FIRST.

Another limitation arises when an entity tries to re-submit a pending transaction created to interact with the FIRST protected protocol, perhaps with a higher gas fee. Since the transaction is seen in the pending pool by all the entities, it increases the chances of getting frontrun. Lastly, our framework does not support the interaction of two FIRST protected contracts, which we aim to address in future work. We note that FIRST is a probabilistic solution as it recommends a fee to be paid by the users in the system to avoid getting frontrun with a high probability. However, as specified by the advantage statement in our theoretical analysis, there is a chance that a sufficiently funded and powerful adversary can outpace and frontrun honest users. To successfully frontrun a target user, an adversary not only needs commensurately larger computational resources than the norm to compute the VDF proof faster, but the adversary also needs to delay the target user's transaction in the mempool for the duration of time it takes to compute a valid VDF proof by inserting other transactions with higher fees than the target. This is a high barrier even for a very resourceful adversary. This is the price we pay for having an autonomous distributed system with no central control. Achieving zero frontrunning probability would require centralized transaction serialization, compromising blockchain decentralization and scalability.

8 RELATED WORK

Current frontrunning research focuses on attack classification, profitability analysis, and mitigation strategies. We describe each below.

Frontrunning prevention strategies: Research in frontrunning prevention falls into three broad categories: (a) solutions that require direct interaction with miners to include the transaction in the upcoming block (private relayer approaches) [30]; (b) solutions designed for DEXs (protocol incentive design) [43, 44]; and (c) solutions that prevent arbitrary reordering of transactions (order fairness) [23, 24, 26]. In the first category, Alice sends tx_A directly to miners via hidden endpoints to prevent adversaries from identifying her transaction in the pending pool. Flashbots [30] is one example, where entities called *relayers* bundle and forward transactions to miners through private channels. However, relayers themselves could perform frontrunning attacks as they have access to the complete transaction details. The second category of solutions is built for AMM-based DEXs; it reduces the risk of frontrunning by computing an optimal threshold for the frontrunner's transaction and routing the victim's swap request to minimize potential profit extraction. However, these solutions are specific to DEXs and cannot be applied to other dApps such as auctions, naming services, or games [22, 25, 43]. The third category of solutions is built upon the order-fairness property, which ensures that the order of transactions in the finalized block reflects the order in which users submitted them [23, 24, 26]. These solutions require significant changes to the consensus layer, making them impractical, while our approach works with existing EVM-based blockchains without modification.

Surveys of frontrunning and related mechanisms: Eskandari *et al.* presented a taxonomy of frontrunning attacks and analyzed the attack surface of top dApps [22]. Qin *et al.* [32] extended the taxonomy of [22] and quantified the profit made by blockchain extractable value [32]. Daian *et al.* [20] revealed frontrunning bots competing in priority gas auctions and coined "Miner Extractable Value" (MEV) to describe miners reordering transactions for profit. [12] presented the state-of-the-art in frontrunning research and proposed a categorization of mitigation strategies. Additionally, Yang *et al.* developed a taxonomy of MEV prevention solutions and conducted a comparative analysis of these approaches.

Profitability analysis: The profits made by frontrunners have been quantified by Torres *et al.* [37] and Qin *et al.* [32]; the latter also brought to attention the presence of private transactions submitted to miners. Zhou *et al.* [44] formalized and quantified the profit made by *sandwich attacks* enabled by frontrunning on decentralized exchanges. Qin *et al.* [33] analytically evaluated Ethereum transactions' atomicity, analyzed two flash loan-based attacks, and demonstrated how attackers could have maximized their profit. Wang *et al.* [41] proposed a framework that analyzes the profitability conditions on cyclic arbitrage in DEXs.

There are a couple of prior works [16, 39] that do not fall into any of the aforementioned three categories. LibSubmarine uses a commit-and-reveal scheme to prevent frontrunning [16], where the committer must create a new smart contract for every transaction they submit to a dApp, which is inefficient. In a recent work, Varun *et al.* [39] proposed a machine learning approach to detect transactions that were frontrun in real-time. This approach requires the machine learning model to learn regularly. Further, the approach does not take into account priority fee, hence could fail to identify high priority fee based frontrunning transactions. There are also works that are related to our proposed scheme, such as Slowswap [7], which utilizes VDFs to introduce delays for transactions related to AMMs only. However, the current implementation employs a uniform VDF delay for all transactions, which is not ideal given the dynamic nature of Ethereum. In contrast, FIRST conducts statistical analysis and assigns VDF delay based on the network usage. Another solution in the MEV mitigation space is Radius [8], which also aims to prevent frontrunning and sandwich attacks by implementing encrypted mempools. The Radius solution requires a mempool redesign, limiting its applicability, whereas FIRST integrates seamlessly with any EVM-based blockchain.

9 CONCLUSION

We introduced FIRST, a decentralized framework aimed at mitigating frontrunning attacks on EVM-based smart contracts without necessitating changes to the blockchain consensus layer. Unlike application-specific approaches, FIRST is designed as a versatile and general-purpose solution, ensuring broad applicability across diverse dApps. Experimental results show that FIRST effectively reduces the likelihood of frontrunning attacks on two prominent blockchains: Ethereum and Binance Smart Chain. Additionally, the security guarantees of FIRST are rigorously established through the UC framework. In the future, we will explore gas golfing and better aggregate signature design to help reduce the gas fees needed for on-chain signature verification.

10 ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Award No 2148358, 1914635, 2417062, and the Department of Energy under Award No. DESC0023392. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the Department of Energy.

REFERENCES

[1] dYdX, 2024-7-7. https://dydx.exchange/.
[2] DeFi Lama, 2024-12-08. https://defillama.com/chain/Ethereum.
[3] AAVE, 2024-7-7. https://aave.com/.
[4] Ethereum Name Service, 2024-8-12. https://ens.domains/.
[5] Amazon.eth ENS domain owner disregards 1M USDC buyout offer on OpenSea, 2024-8-12. https://cointelegraph.com/.
[6] More than $10bn in volume has now been traded on OpenSea in 2021, 2024-8-12. https://yahoo.com.
[7] slowswap, 2023-8-12.
[8] theradius, 2023-8-12.
[9] Etherscan, 2024-7-9. https://etherscan.io/chart/tx.
[10] yearn.finance, 2024-7-7. https://yearn.finance/.
[11] Compound Finance, 2024-8-12. https://compound.finance/.
[12] Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. Sok: Mitigation of front-running in decentralized finance. In *International Conference on Financial Cryptography and Data Security*, pages 250–271. Springer, 2022.
[13] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual International Cryptology Conference*. Springer, 2018.
[14] Dan Boneh, Benedikt Bünz, and Ben Fisch. A Survey of Two Verifiable Delay Functions. *IACR Cryptol. ePrint Arch.*, 2018.
[15] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International conference on the theory and applications of cryptographic techniques*. Springer, 2003.
[16] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts. In *27th USENIX Security Symposium*, 2018.
[17] Ran Canetti. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, 2004.
[18] Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. A study of inline assembly in solidity smart contracts. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1123–1149, 2022.
[19] Bram Cohen and Krzysztof Pietrzak. The chia network blockchain, 2019.
[20] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *IEEE Symposium on Security and Privacy (SP)*, 2020.
[21] Karim Eldefrawy, Sashidhar Jakkamsetti, Ben Terner, and Moti Yung. Standard model time-lock puzzles: Defining security and constructing via composition. Cryptology ePrint Archive, Paper 2023/439, 2023. https://eprint.iacr.org/2023/439.
[22] Shayan Eskandari, Mahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. *Financial Cryptography*, 2019.
[23] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. In *Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop*, 2022.
[24] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 475–489, 2023.
[25] Rami Khalil, Arthur Gervais, and Guillaume Felley. TEX-A Securely Scalable Trustless Exchange. *IACR Cryptol. ePrint Arch.*, 2019.
[26] Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020.
[27] Esteban Landerreche, Marc Stevens, and Christian Schaffner. Non-interactive cryptographic timestamping based on verifiable delay functions. In *International Conference on Financial Cryptography and Data Security*. Springer, 2020.
[28] Craig McCann. Detecting Personal Trading Abuses, 2000. https://www.slcg.com/.
[29] Michael Mirkin, Yan Ji, Jonathan Pang, Ariah Klages-Mundt, Ittay Eyal, and Ari Juels. BDoS: Blockchain Denial-of-Service. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM SIGSAC Conference on Computer and Communications Security*, 2020.
[30] Alex Obadia. Flashbots: Frontrunning the MEV Crisis, 2024-7-7. https://medium.com/flashbots/frontrunning-the-mev-crisis-40629a613752.
[31] Krzysztof Pietrzak. Simple verifiable delay functions. In *Innovations in theoretical computer science conference (ITCS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
[32] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.
[33] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. Attacking the defi ecosystem with flash loans for fun and profit. In *International Conference on Financial Cryptography and Data Security*. Springer, 2021.
[34] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. *Massachusetts Institute of Technology. Laboratory for Computer Science*, 1996.
[35] solidproof. solidproof, 2024-7-7. https://solidproof.io/kyc.
[36] EigenLayer Team. Eigenlayer: The restaking collective, 2024-7-7. https://docs.eigenlayer.xyz/eigenlayer/overview/whitepaper.
[37] Christof Ferreira Torres, Ramiro Camino, et al. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the Ethereum blockchain. In *30th USENIX Security Symposium*, 2021.
[38] Uniswap. Uniswap, 2024-8-12. https://uniswap.org/.
[39] Maddipati Varun, Balaji Palanisamy, and Shamik Sural. Mitigating Frontrunning Attacks in Ethereum. In *Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure*, 2022.
[40] vbuterin. EIP 1559 FAQ, 2024-7-7. https://notes.ethereum.org/@vbuterin/eip-1559-faq.
[41] Ye Wang, Yan Chen, Haotian Wu, Liyi Zhou, Shuiguang Deng, and Roger Wattenhofer. Cyclic Arbitrage in Decentralized Exchanges. *Available at SSRN 3834535*, 2022.
[42] Benjamin Wesolowski. Efficient verifiable delay functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2019.
[43] Liyi Zhou, Kaihua Qin, and Arthur Gervais. A2MM: Mitigating Frontrunning, Transaction Reordering and Consensus Instability in Decentralized Exchanges. *arXiv preprint arXiv:2106.07371*, 2021.
[44] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-frequency trading on decentralized on-chain exchanges. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

A DEFINITIONS OF CRYPTOGRAPHIC PRIMITIVES

Definition A.1. *(Verifiable Delay Function [13]) A verifiable delay function, $\mathcal{V}$ is defined over three polynomial time algorithms. (a)* $\mathsf{Setup}(\lambda, T) \rightarrow pp = (ek, vk)$*: This is a randomized algorithm that takes a security parameter λ and a desired puzzle difficulty T and produces public parameters pp that consists of an evaluation key ek and a verification key vk. We require* Setup *to be polynomial-time in λ. By convention, the public parameters specify an input space X and an output space Y. We assume that X is efficiently sampleable.* Setup *might need secret randomness, leading to a scheme requiring a trusted setup. For meaningful security, the puzzle difficulty T is restricted to be sub-exponentially sized in λ. (b)* $\mathsf{Eval}(ek, x) \rightarrow (y, \pi)$*: This algorithm takes an input $x \in X$ and produces an output $y \in Y$ and a (possibly empty) proof π.* Eval *may use random bits to generate the proof π but not to compute y. For all pp generated by* $\mathsf{Setup}(\lambda, T)$ *and all $x \in X$, algorithm* $\mathsf{Eval}(ek, x)$ *must run in parallel time T with poly$(\log(T), \lambda)$ processors.*

(c) $\mathsf{Verify}(vk, x, y, \pi) \rightarrow$ *{"accept", "reject"}: This is a deterministic algorithm that takes an input, output and proof and outputs accept or reject. The algorithm must run in total time polynomial in* $\log T$ *and λ. Notice that* Verify *is much faster than* Eval.

Definition A.2. *(Aggregate signature [15]) An aggregate signature scheme is defined over five polynomial time algorithms: (*KeyGen, Sign, Verify, Aggregate, AggregateVerification*). Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two multiplicative cyclic groups of prime order p generated by g_1 and g_2, respectively. Let $\mathbb{U}$ be the universe of users.* $\mathsf{KeyGen}(1^\lambda) \rightarrow (x_i, v_i)$*: Each user picks random $x_i \leftarrow \mathbb{Z}_p$ and does $v_i \leftarrow g_2^{x_i}$. The*

user's public key is $v_i \in \mathbb{G}_2$ and secret key is $x_i \in \mathbb{Z}_p$.
$\mathsf{Sign}(x_i, M_i) \rightarrow \sigma_i$*: Each user $i \in \mathbb{U}$, given their secret key x_i and message of their choice M_i computes hash $h_i \leftarrow H(M_i)$ and signs $\sigma_i \leftarrow h_i^{x_i}$ where $\sigma_i \in \mathbb{G}_1$ and $H : \{0,1\}^* \rightarrow \mathbb{G}_1$.*
$\mathsf{Verify}(v_i, M_i, \sigma_i) \rightarrow \{\mathsf{true}, \mathsf{false}\}$*: Given public key v_i of user i, a message M_i and σ_i, compute $h_i \leftarrow H(M_i)$ and return true if $e(\sigma_i, g_2) = e(h_i, v_i)$.*
$\mathsf{Aggregate}(M_1, \ldots, M_n, \sigma_1, \ldots, \sigma_n) \rightarrow \sigma_{\mathsf{agg}}$*: Given each user i's signature σ_i on a message of their choice M_i, compute $\sigma_{agg} \leftarrow \prod_{i=1}^{n} \sigma_i$ where $n = |U|$.*
$\mathsf{AggregateVerification}(\sigma_{\mathsf{agg}}, M_1, \ldots, M_n, pk_1, \ldots, pk_n) \rightarrow \{\mathsf{true}, \mathsf{false}\}$*: To verify aggregated signature σ_{agg}, given original messages M_i along with the respective signing users' public keys v_i, check if:*

(1) All messages M_i are distinct, and;
(2) For each user $i \in U$, $e(\sigma_i, g_2) = \prod_{i=1}^{n} e(h_i, v_i)$ holds true where $h_i \leftarrow H(M_i)$.

B EIP 1559

The London hard fork to Ethereum introduces novel transaction pricing mechanisms to improve the predictability of gas prices even during dynamic periods [40]. Users are now required to pay a *base fee*, which is a fee computed according to a formula that may increase or decrease per block depending on network utilization. Besides a base fee, a user is encouraged to pay a *priority fee* to incentivize the validators to prioritize the user's transactions. The transactions that follow EIP-1559 are termed *Type-2* transactions. While Ethereum has adopted EIP-1559, it's worth noting that other prominent blockchain networks, like Binance Smart Chain, have not yet implemented this standard. Despite the differing approaches, Ethereum and Binance Smart Chain remain two of the most widely used blockchain platforms. In our evaluation, we leverage these platforms as references to evaluate the proposed framework and demonstrate its applicability.

C UC FUNCTIONALITIES

C.1 Proof of Theorem 5.1

We assume the existence of eight tables: uTable, aTable, cTable, sTable, idTable, scTable, bcTable and txpoolTable that store the internal state of $\mathcal{F}_{\text{FIRST}}$ and are accessible at any time by $\mathcal{F}_{\text{setup}}$ (Figure 8), $\mathcal{F}_{\text{bc}}$ (Figure 3), and $\mathcal{F}_{\text{construct}}$ (Figure 4), which are time-synchronized functionalities. The uTable is used to store user transaction specific information, aTable is used to store the signatures of verifiers and users, cTable keeps track of VDF-specific challenges issued to users, sTable stores the aggregated signatures of verifiers, and idTable stores the identifiers and keys of users. The scTable stores the deployed smart contract address and code, bcTable stores the generated transactions, and txpoolTable stores the current transaction pool. We assume that $\mathcal{F}_{\text{setup}}$'s t_1 and t_2 time period verification implicitly checks that t_1 and t_2 are in the same unit of time (i.e., both are in seconds, minutes, etc.).

We note that $\mathcal{F}_{\text{bc}}$ does not completely follow EIP-1559 because Ethereum, like other real-world protocols and systems, is constantly evolving, and as these systems change the ideal world would need to be constantly updated to model the real world accurately. $\mathcal{F}_{\text{bc}}$ incorporates block size based on maximum fees per block and the block hash rate, and is still general enough to model even non-EIP-1559 blockchains similar to the real world *FIRST* protocol which is applicable to multiple blockchain types.

To make the presentation clear, for each corruption case, through a complete run of the protocol, we discuss the two worlds separately,

Functionality $\mathcal{F}_{\text{setup}}$

Setup: On receiving tuple $(\text{setup}, t_1, t_2, d, k, \lambda, \alpha, sid)$ from dApp creator dAC, $\mathcal{F}_{\text{setup}}$ verifies that $t_1 > t_2$, if not return $\perp$. $\mathcal{F}_{\text{setup}}$ sets value of VDF delay to t_1 (s in $\mathcal{F}_{\text{vdf}}^{\gamma}$), locally stores variables d, k, λ, α and initializes *FIRST* recommended fee $FIRST\_FEE = 0$.

KeyGen: Upon receiving a request $(\text{KeyGen}, uid, sid)$ from user u, $\mathcal{F}_{\text{setup}}$ calls $\mathcal{F}_{\text{sig}}$ with (KeyGen, uid). When $\mathcal{F}_{\text{sig}}$ returns $(\text{VerificationKey}, uid, pk_u)$, $\mathcal{F}_{\text{setup}}$ records the pair (u, pk_u) in idTable and returns $(\text{VerificationKey}, uid, pk_u)$ to the user and $\mathcal{S}$.

Sign: When $\mathcal{F}_{\text{setup}}$ receives a request $(\text{Sign}, uid, m, sid)$ from user u, it forwards the request to $\mathcal{F}_{\text{sig}}$, who returns $\{(\text{Signature}, uid, m, \sigma), \perp\}$. If return value is not $\perp$, $\mathcal{F}_{\text{setup}}$ stores $(m, \sigma, pk_u, 1)$ in aTable, where pk_u is u's verification key created and stored during key generation. $\mathcal{F}_{\text{setup}}$ forwards $(\text{Signature}, uid, m, \sigma)$ to u and $\mathcal{S}$, else returns $\perp$ to both.

Verify: When $\mathcal{F}_{\text{setup}}$ receives $(\text{Verify}, uid, m, \sigma, pk', sid)$ from user u, it forwards the request to $\mathcal{F}_{\text{sig}}$, who returns $(\text{Verified}, uid, m, f)$, $f \in \{0, 1, \phi\}$. $\mathcal{F}_{\text{setup}}$ records (m, σ, pk', f) in aTable and returns $(\text{Verified}, uid, m, f)$ to both user and $\mathcal{S}$. **Aggregate Signature**: Upon receiving $(\text{Aggregate}, M_1, \ldots M_n, pk_1, \ldots, pk_n, \sigma_{V_1} \ldots \sigma_{V_n}, sid)$ from C, $\mathcal{F}_{\text{setup}}$ checks if a tuple $(\sigma_{\text{agg}}, M_1 \ldots M_n, pk_1, \ldots, \text{pk}_n)$ already exists in sTable, if so, it forwards $(\text{Aggregated}, \sigma_{\text{agg}})$ to C and $\mathcal{S}$. Else, it checks if $n > |\mathbb{V}|/2$, if not then $\perp$ is returned to C and $\mathcal{S}$. If previous check passed, $\mathcal{F}_{\text{setup}}$ generates a string $\sigma_{\text{agg}} \leftarrow\$ \{0,1\}^{\lambda}$, adds $(\sigma_{\text{agg}}, M_1, \ldots M_n, pk_1, \ldots, pk_n)$ to table sTable, and forwards $(\text{Aggregated}, \sigma_{\text{agg}})$ to C and $\mathcal{S}$.

Aggregate Verify: Upon receiving $(\text{aggVer}, \sigma_{\text{agg}}, M_1, \ldots M_n, pk_1, \ldots, pk_n, sid)$ from an entity, $\mathcal{F}_{\text{setup}}$ checks if tuple $(\sigma_{\text{agg}}, M_1, \ldots M_n, pk_1, \ldots, pk_n)$ exists in sTable. If yes, it forwards "accept", else forward "reject" to the calling entity and $\mathcal{S}$.

Hash Interface: On receiving a message $(hash, m, sid)$ from a user u, $\mathcal{F}_{\text{setup}}$ checks if tuple (m, h) exists in hashTable. If so, it returns h and exits. If not then $\mathcal{F}_{\text{setup}}$ creates $h \leftarrow\$ \{0,1\}^{\lambda}$, adds (m, h) to hashTable, and returns h to u.

Calculate FIRST Fee: For every new block mined, $\mathcal{F}_{\text{setup}}$ sends getData() request to $\mathcal{F}_{\text{bc}}$. $\mathcal{F}_{\text{setup}}$ then checks priority fee (f_i where $i \in [1 \ldots n]$) paid by $tx_1 \ldots tx_n$ transactions in the latest block that waited less than t_2 time, and calculates the value of $f_{avg} = 1/n \times \sum f_i$. $FIRST\_FEE = \alpha \times f_{avg} + (1 - \alpha) \times FIRST\_FEE$.

Return FIRST fee: On receiving request $(returnFee, sid)$ from user, $\mathcal{F}_{\text{setup}}$ returns current value of *FIRST_FEE*.

Figure 8: Ideal functionality for system setup and signatures.

and show that $\mathcal{Z}$'s view will be the same. **Part 1**: Let us first consider the system and parameter setup described in Protocols 1, 2, and 3 . $\mathcal{Z}$ initializes $\mathcal{F}_{\text{bc}}$ with $(init, sid, \mathbb{P}, \mathbb{H})$.

1) Case 0: All verifiers are honest

a) **Real-world**: In the real-world (Protocols 1, 2, 3), the *dAC* generates a smart contract, deploys it on the BC, and initializes *FIRST_FEE* calculation. The n verifiers will generate their keypairs, $(pk_i, sk_i), i \in [1..n]$. $\mathcal{Z}$ sees the *SC* and each verifier's pk. *dAC* will pick a T, and initialize the $\mathcal{V}$ class group with a negative prime d. Note that since all verifiers are honest, $\mathcal{Z}$ does not get to see their internal state, and secret keys. The view of $\mathcal{Z}$ will be $(SC, pp = (\mathbb{G}, T, d), pk_1, \ldots, pk_n, \lambda, k, t_1, t_2, \alpha$,FIRST_FEE), where λ is the security parameter, and k is the multiplying factor for t_2.

b) **Ideal-world**: $\mathcal{S}$ picks a security parameter λ, $(T, k) \leftarrow\$ \mathbb{Z}^+$, negative prime d, vdf delay t_1, target mempool wait time t_2, and EWMA parameter value α, and sends $(\text{setup}, t_1, t_2, d, k, \alpha)$ to $\mathcal{F}_{\text{setup}}$ to start the *FIRST_FEE* calculation. $\mathcal{S}$ calls $\mathcal{F}_{\text{bc}}$ with $(sid, \text{deploy}, SC.id, code)$ (this step is implicit in all the following game hybrids). $\mathcal{A}$ sends getData() to $\mathcal{F}_{\text{bc}}$ to get a copy of *SC* (also including all contents of the blockchain). $\mathcal{S}$ makes n calls to $\mathcal{F}_{\text{setup}}$, $(\text{KeyGen}, vid_i), i \in [1..n]$. $\mathcal{F}_{\text{setup}}$ returns $(\text{VerificationKey}, vid_i, pk_i)$ to $\mathcal{S}$. $\mathcal{S}$ generates a random $\mathbb{G}$ such that $\mathbb{G} = Cl(d)$, and sets $pp = (\mathbb{G}, T, d)$. $\mathcal{S}$ call **Return FIRST fee** to get computed value of *FIRST_FEE*. Thus the view of $\mathcal{Z}$ is the same as the real-world. The view of $\mathcal{Z}$ will be $(SC, pp = (\mathbb{G}, T, d), pk_1, \ldots, pk_n, \lambda, k, t_1, t_2, \alpha, FIRST\_FEE)$.

2) Case 1: Some verifiers are corrupted

a) **Real-world**: Per our adversary model, less than half of the verifiers can be corrupted. *dAC* deploys the SC on the blockchain, initializes *FIRST_FEE* calculation, and all verifiers will generate their keypairs. In this case, $\mathcal{Z}$ will have access to both pk and sk of a corrupted verifier. $\mathcal{Z}$ will also have access to the corrupted verifiers' $D_i = U_i = \emptyset$. Verifiers, corrupt or otherwise, have no role to play in Protocol 3. *dAC* will deploy the *SC* on the blockchain as before, and will generate $\mathbb{G} = Cl(d), T$. Let the set of corrupted verifiers be $\mathbb{V}'$, such that $\mathbb{V}' \subset \mathbb{V}$, and $|\mathbb{V}'| < |\mathbb{V}|/2$. The view of $\mathcal{Z}$ will be $(SC, pp = (\mathbb{G}, T, d), \{pk_i, sk_i, D_i, U_i\}_{i \in \mathbb{V}'}, \{pk_j\}_{j \in (\mathbb{V})}, \lambda, k, t_1, t_2, \alpha, FIRST\_FEE)$.

b) **Ideal-world**: As in Case 0, $\mathcal{S}$ simulates *dAC*'s role and receives from $\mathcal{F}_{\text{setup}}$ $(\text{Init}, T, d, FIRST\_FEE)$. $\mathcal{S}$ calls $\mathcal{F}_{\text{bc}}$ with $(\text{deploy}, SC.id, code)$. $\mathcal{Z}$ sends getData() to $\mathcal{F}_{\text{bc}}$ to get a copy of *SC* (also including all contents of the blockchain). For the honest verifiers, $\mathbb{V} - \mathbb{V}'$, $\mathcal{S}$ creates $pk \leftarrow\$ \{0,1\}^\lambda$. Corrupt verifiers in $\mathbb{V}' \subset \mathbb{V}$ are handled by $\mathcal{Z}$. Following the same procedure as in Case 0's ideal world, $\mathcal{S}$ generates a random $\mathbb{G}$ s.t., $\mathbb{G} = Cl(d)$ and outputs $(SC, pp = (\mathbb{G}, T, d), pk_1, \ldots, pk_n, k)$. The view of $\mathcal{Z}$, taking into account the additional information $\mathcal{Z}$ has from corrupted verifiers will be $(SC, pp = (\mathbb{G}, T, d), \{pk_i, sk_i, D_i, U_i\}_{i \in \mathbb{V}'}, \{pk_j\}_{j \in \mathbb{V}}, \lambda, k, t_1, t_2, \alpha, FIRST\_FEE)$, which is the same as the real-world.

Part 2: Now, let us consider Alice's setup as given in Protocol 4.

1) Case 0: Alice and all verifiers are both honest

a) **Real world**: Alice generates M_A, hashes it, signs the digest, h: $\sigma_A \leftarrow \text{Sign}(h, sk_A)$, and sends (h, σ_A) to all members of $\mathbb{V}$. $\mathcal{Z}$'s view will be $\emptyset$ (since all verifiers are honest, it does not have access to their inputs).

b) **Ideal world**: $\mathcal{S}$ simulates Alice and will receive (req, aliceID, h, σ_A) from $\mathcal{F}_{\text{construct}}$. $\mathcal{S}$ does not take any further actions.

2) Case 1: Alice is honest, some verifiers are corrupt

a) **Real world**: Alice generates the (M_A, h, σ_A) as in Case 0, and sends (h, σ_A) to $\mathbb{V}$. If $\mathbb{V}'$ is the set of corrupted verifiers, $\mathcal{Z}$'s view will consist of $\mathbb{V}'$'s inputs, i.e., $(\{sk_i\}_{i \in \mathbb{V}'}, h, \sigma_A)$.

b) **Ideal world**: $\mathcal{S}$ generates an $h \leftarrow\$ \{0,1\}^k$ (note that verifiers do not know the preimage). $\mathcal{S}$ then calls $\mathcal{F}_{\text{setup}}$ with (Sign, aid, h), where aid is chosen at random. $\mathcal{F}_{\text{setup}}$ returns $(\text{Signature}, aid, h, \sigma_{aid})$. $\mathcal{S}$ outputs (h, σ_{aid}).

3) Case 2: Alice is corrupt and all verifiers are honest

a) **Real world**: Alice generates M_A and σ_A over M_A's digest. If the signature does not verify, verifiers will eventually return $\perp$. If Alice does not send anything, verifiers will do nothing. In any case, $\mathcal{Z}$'s view will be $(M_A = (addr_A, f_{\text{name}}, addr_{SC}), h, \sigma_A)$.

b) **Ideal world**: $\mathcal{S}$ gets (h, σ_A) from $\mathcal{Z}$. $\mathcal{S}$ does not take any further actions. $\mathcal{Z}$'s view will be $(M_A = (addr_A, f_{\text{name}}, addr_{SC}), h, \sigma_A)$.

4) Case 3: Both, Alice and some verifiers are corrupt Note that this cannot be locally handled by $\mathcal{Z}$, as one might expect, since some verifiers are still honest.

a) **Real world**: Alice's actions will be the same as in Case 2's real world. $\mathcal{Z}$'s view will be $(M_A, h, \sigma_A, \{sk_i\}_{i \in \mathbb{V}'})$, where $\mathbb{V}'$ is the set of corrupted verifiers.

b) **Ideal world**: $\mathcal{S}$ gets (h, σ_A) from $\mathcal{Z}$. $\mathcal{S}$ does not take any further actions. $\mathcal{Z}$'s view is same as real world.

Part 3: Now let us consider Alice, C, and verifiers' interaction as given in Protocol 6, 5, and 7. In the following cases, whenever some verifiers $(\mathbb{V}')$ are corrupt, $|\mathbb{V}'| < |\mathbb{V}|/2$, hence, a majority of verifiers are still honest.

1) Case 0: Alice, C and all verifiers are honest

a) **Real-world**: On receiving a new VDF request from Alice, C picks an $l \leftarrow\$ \mathbb{P}$, all verifiers send (M_i, σ_{V_i}) to C. C will verify the signatures, and will return the aggregate signature σ_{agg} to Alice, who will then compute the VDF proof, (π, y). This proof is sent to the $\mathbb{V}$ who will verify it before submitting their signatures to the C for aggregation. The aggregated signature is sent to Alice by the C who verifies it, and eventually submits tx_A with the current *FIRST_FEE* to the *SC* using $(sid, \text{invoke}, tx_A)$. $\mathcal{Z}$'s view will only be $\{pk_i\}_{i \in \mathbb{V}}$ initially, and it will see tx_A only when it hits the transaction pool.

b) **Ideal-world**: $\mathcal{S}$ creates M_A, creates hash $h_A = H(M_A)$ and calls $\mathcal{F}_{\text{setup}}$ and gets σ_A. It then forwards $(\text{req}, h_A, \sigma_A)$ to $\mathcal{F}_{\text{construct}}$'s **User request** function and receives "success" and (req, aliceID, h_A, σ_A). $\mathcal{S}$ generates l on behalf of the C and sends $(l, \text{aliceID})$ to $\mathcal{F}_{\text{construct}}$ using **Coordinator request** function and it receives $(\text{valid}, l, h_A, \sigma_A, pk_A)$. For each V_i, $\mathcal{S}$ signs the $M_i = (l, h_A, \mathsf{V}_i, block_{curr})$ using $\mathcal{F}_{\text{setup}}$ and $block_{curr}$ is retrieved by calling $\mathcal{F}_{\text{bc}}$, i.e., using getBlockNum() and being returned $block_{curr} \leftarrow \lceil num_{tx}/block_{qty} \rceil$. $\mathcal{S}$ sends $(\mathsf{V}_i, M_i, \sigma_{M_i})$ to $\mathcal{F}_{\text{construct}}$ using **Coordinator response** function call. $\mathcal{S}$ then simulates the aggregation step of C by calling $\mathcal{F}_{\text{setup}}$ and receives σ_{agg}. $\mathcal{S}$ calls $\mathcal{F}_{\text{construct}}$ **User response** to send σ_{agg} to Alice. $\mathcal{S}$ calls $\mathcal{F}_{\text{vdf}}$ (start, l) function to start $\mathcal{V}$ delay. After $\mathcal{V}$ delay time, $\mathcal{S}$ calls (output, l) function in $\mathcal{F}_{\text{vdf}}$ to generate the $\mathcal{V}$ proof which returns (s, p). $\mathcal{S}$ then verifies the proof calling (verify, l, p, s) on behalf of each V_i and generates M'_i and σ'_{V_i} in a straightforward way. $\mathcal{S}$ aggregates all the signatures from $\mathbb{V}$ using $\mathcal{F}_{\text{setup}}$ and generates σ'_{agg} and forwarded to Alice using $\mathcal{F}_{\text{construct}}$'s **User response** function. $\mathcal{S}$ creates $tx_A = (\sigma'_A, M', pk_A, FIRST\_FEE)$, where $M' = (M_A, M_{\text{agg}}, M'_{\text{agg}}), M'_{\text{agg}} = (\sigma'_{\text{agg}}, M'_1, \ldots, M'_n, pk_{\mathsf{V}_1}, \ldots, pk_{\mathsf{V}_n}), M_{\text{agg}} =$

$(\sigma_{\text{agg}}, M_1, \ldots, M_n, pk_{\mathbb{V}_1} \ldots pk_{\mathbb{V}_n})$, and *FIRST_FEE* is retrieved by sending (*returnFee*) request to $\mathcal{F}_{\text{setup}}$. $\mathcal{S}$ sends $(sid, \text{invoke}, tx_A)$ to $\mathcal{F}_{\text{bc}}$ calling smart contract with *SC.id*. tx_A will get added to the txpoolTable. While the tx_A is in the pending pool, the adversary can try to delay tx_A from being mined by submitting transactions with higher fees than tx_A, while the adversary generates a valid $\mathcal{V}$ proof. We point out that the adversary will need to submit enough transactions with higher fees so that tx_A does not appear in any of the blocks before adversary has a valid $\mathcal{V}$ proof, which would require an exorbitant amount of fees just like in the real world. When tx_A eventually gets mined and added to a block, it will appear in bcTable with the corresponding blockNum and if an adversary's valid transaction did not get mined before Alice's then Alice did not get frontrunned (this step is implicit in all the following game hybrids). The *code* associated with *SC.id* checks that the σ_{agg} and σ'_{agg} are signed by majority $\mathbb{V}$, $block_{curr}$ signed in σ_{agg} is valid, and that the $h_A \stackrel{?}{=} H(M_A)$. If any of the checks fail, the smart contract returns $\perp$, else outputs a valid transaction tx'.

2) Case 1: Alice, C are honest, some verifiers are corrupt

a) **Real-world**: Honest C generates $l \leftarrow\!\$\ \mathbb{P}$. Corrupted verifiers can either: 1) deliberately fail Alice's signature verification (Step 4 of Protocol 6), or 2) create a bogus signature over a possibly incorrect message (Steps 5, 6 of Protocol 6). In both cases, the corrupt verifiers in $\mathbb{V}'$ will not contribute towards σ_{agg}, since the C needs a majority to abort the process and return $\perp$ (Step 18 of Protocol 6). As long as we have a honest majority in $\mathbb{V}$, honest C will create and return σ_{agg} to Alice, who will then evaluate the VDF and generate (π, y), and send (π, y) to all members of $\mathbb{V}$. Similarly, during the generation of σ'_{agg}, C can ignore the inputs from $\mathbb{V}'$. C sends $(M_{\text{agg}}, block_{curr})$ to Alice. Honest Alice eventually outputs tx_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, \pi, y, tx_A, block_{curr}, pp, t_1, t_2, \alpha)$. [8]

b) **Ideal-world**: $\mathcal{S}$ needs to simulate the actions of C and Alice to $\mathcal{Z}$. $\mathcal{S}$ creates M_A, creates hash $h_A = H(M_A)$ and calls $\mathcal{F}_{\text{setup}}$ and gets σ_A. It then forwards $(\text{req}, h_A, \sigma_A)$ to $\mathcal{F}_{\text{construct}}$ and receives "success" and $(\text{req}, \text{aliceID}, h_A, \sigma_A)$. $\mathcal{S}$ picks an $l \leftarrow\!\$\ \mathbb{P}$, calls **Coordinator request** function in $\mathcal{F}_{\text{construct}}$, and sends l to $\mathcal{Z}$. If members of $\mathbb{V}'$ return $\perp$ for Alice's signature verification or return bogus signatures from $\mathbb{V}'$ ($\mathcal{S}$ can check these using **Verify** function call in $\mathcal{F}_{\text{setup}}$), $\mathcal{S}$ ignores them, since $|\mathbb{V}'| < |\mathbb{V}|/2$. $\mathcal{S}$ then calls $\mathcal{F}_{\text{setup}}$'s **Aggregate Signature** function with (Aggregate, $M_1, \ldots, M_n, pk_1, \ldots, pk_n, \sigma_{\mathbb{V}_1}, \ldots, \sigma_{\mathbb{V}_n}$) to aggregate all honest majority $\mathbb{V}$'s signatures. $\mathcal{F}_{\text{setup}}$ returns σ_{agg} to $\mathcal{S}$. $\mathcal{S}$ then calls $\mathcal{F}_{\text{vdf}}$ to generate $\mathcal{V}$ proof (s, p). $\mathcal{S}$ sends (l, p, s) to $\mathcal{Z}$. Members of $\mathbb{V}'$ will send $\{\sigma'_i\}_{i \in \mathbb{V}'}$ to $\mathcal{S}$. $\mathcal{S}$ will simulate signatures for members of $(\mathbb{V} - \mathbb{V}')$ in a straightforward way and calls $\mathcal{F}_{\text{bc}}$'s getBlockNum() function to get the $block_{curr} \leftarrow \lceil num_{tx}/block_{qty} \rceil$ value. $\mathcal{S}$ generates the σ'_{agg} similar to the previous σ_{agg}, by taking the majority signatures. Since members of $\mathbb{V}'$ will be in a minority, even if they return $\perp$, it will not affect the creation of σ'_{agg}. Finally $\mathcal{S}$ generates Alice's signature over M', creates tx_A and submits it to $\mathcal{F}_{\text{bc}}$. The view of $\mathcal{Z}$, who controls $\mathbb{V}'$ will be $(l, \sigma_A, h_A, pk_A, p, s, tx_A, block_{curr}, pp, t_1, t_2, \alpha)$

3) Case 2: Alice, all verifiers are honest, C is corrupt

a) **Real-world**: On receiving VDF request from Alice, corrupt C could either pick $l \leftarrow\!\$\ \mathbb{P}$ which has already been assigned to another user or an $l \notin \mathbb{P}$, in this case the honest members of $\mathbb{V}$ identifying the C as corrupt, will not generate an accept message which can be aggregated by the C and the C cannot proceed. If the C had picked $l \leftarrow\!\$\ \mathbb{P}$ correctly, on receiving the accept messages and signatures from $\mathbb{V}$, C can still choose to create σ_{agg} that would fail verification, in this case Alice's checks would fail, identifying the C as corrupt and she would not proceed further with the protocol. If the C had created σ_{agg} correctly, Alice would generate the $\mathcal{V}$ proof and send it for verification to all $\mathbb{V}$. Upon verification, $\mathbb{V}$ send their replies to C for aggregation. Like the previous aggregation step, if the C creates a corrupt message in this step, Alice would be able to identify the C as malicious. If the C sends Alice a valid σ'_{agg}, Alice eventually outputs tx_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, \pi, y, tx_A, block_{curr}, pp, t_1, t_2, \alpha\ , \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \ldots |\mathbb{V}|]$.

b) **Ideal-world**: $\mathcal{S}$ needs to simulate the actions of $\mathbb{V}$ and Alice to $\mathcal{Z}$. $\mathcal{Z}$ picks an $l \leftarrow\!\$\ \mathbb{P}$, and sends to $\mathcal{S}$. $\mathcal{S}$ sends $(l, \text{aliceID})$ to $\mathcal{F}_{\text{construct}}$ and if it received (used, l) then l has been used before and $\mathcal{S}$ would return $\perp$ stopping the protocol. If $\mathcal{Z}$ picked a valid l, $\mathcal{S}$ simulates the operations of the honest $\mathbb{V}$. $\mathcal{S}$ sends each V_i's accept message to $\mathcal{Z}$ who creates an σ_{agg} by calling $\mathcal{F}_{\text{setup}}$'s Aggregate Signature function call. $\mathcal{S}$ verifies σ_{agg} before proceeding, else return $\perp$. This is sent to $\mathcal{S}$ who simulates Alice's operation of computing the VDF, before simulating the members of $\mathbb{V}$'s response accepting Alice's VDF proof computation, and forwarding $(M'_1, \ldots M'_n, pk_1, \ldots, pk_n, \sigma'_{V_1} \ldots \sigma'_{V_n})$ to $\mathcal{Z}$. If $\mathcal{Z}$ does not aggregate the signatures from $\mathbb{V}$ correctly, and sends corrupted/malformed σ'_{agg} to $\mathcal{S}$, the signature verification by $\mathcal{S}$ would fail. Finally $\mathcal{S}$ simulates Alice's signature over tx_A and submits to $\mathcal{F}_{\text{bc}}$, i.e., $(sid, \text{invoke}, tx_A)$. The view of $\mathcal{Z}$, who controls C will be $(l, \sigma_A, h, pk_A, p, s, tx_A, block_{curr}, pp,\ t_1, t_2, \alpha,\ \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \ldots |\mathbb{V}|]$.

4) Case 3: Alice is honest, C and some verifiers are corrupt

a) **Real-world**: On receiving VDF request from Alice, corrupt C could pick $l \leftarrow\!\$\ \mathbb{P}$ which has already been assigned to another user or an $l \notin \mathbb{P}$, in this case the honest majority of $\mathbb{V} - \mathbb{V}'$ would not generate a signature for C identifying the C. The corrupt $\mathbb{V}'$ could choose to generate accept messages and send them to C. The C can create σ_{agg} using the corrupt $\mathbb{V}'$'s accept messages but this would fail verification on when Alice receives σ_{agg} as $|\mathbb{V}'| < |\mathbb{V}|/2$. If the C had picked $l \leftarrow\!\$\ \mathbb{P}$ correctly, on receiving the accept messages and signatures from $\mathbb{V}$, C can still choose to create σ_{agg} that would fail verification, in this case Alice's checks would fail, identifying the C as corrupt and she would not proceed further with the protocol. If the C had created σ_{agg} correctly, Alice would generate the $\mathcal{V}$ proof and send it for verification to all $\mathbb{V}$. Upon verification, honest members $\mathbb{V} - \mathbb{V}'$, send their replies to C for aggregation. The dishonest members $\mathbb{V}'$ could either choose to not send a valid "*accept*" message for aggregation or choose to send a corrupt message for aggregation. The C could choose to create a corrupt message by aggregating less than $|\mathbb{V}|/2$ messages or create a junk σ'_{agg}. Like the previous aggregation step, if the C creates a corrupt σ'_{agg} in this step, Alice would be able to identify the C as malicious because of the checks she does on receiving the messages from C.

[8] We note that $\mathcal{Z}$ always has access to all the *BC* data: In the real world $\mathcal{Z}$ can query a full node, run a light node, etc. In the ideal world, $\mathcal{Z}$ can send a getData() request to $\mathcal{F}_{\text{bc}}$. Without loss of generality, we say the $\mathcal{Z}$'s view includes $block_{curr}$ because a given $block_{curr}$ is only tied to the current request and signifies the current block number on the *BC* when the VDF request was received by the verifiers.

If the C sends Alice a valid σ'_{agg}, Alice eventually outputs tx_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, \pi, y, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

b) **Ideal-world**: $\mathcal{S}$ needs to simulate the actions of $\mathbb{V} - \mathbb{V}'$ and Alice to $\mathcal{Z}$. $\mathcal{Z}$ picks an $l \leftarrow\$ \mathbb{P}$, and sends to $\mathcal{S}$. If l has been used before, then $\mathcal{S}$ would just return $\perp$ on behalf of honest $\mathbb{V}$. $\mathcal{Z}$ can still choose to create a σ_{agg} with $\mathbb{V}'$ accept messages but when this is sent to $\mathcal{S}$ it would fail verification since $|\mathbb{V}'| < |\mathbb{V}|/2$. If $\mathcal{Z}$ picked a valid l, $\mathcal{S}$ simulates operations of the honest verifiers and sends each $V_i \in \{\mathbb{V} - \mathbb{V}'\}$ accept message to $\mathcal{Z}$ who creates an σ_{agg}. This is sent to $\mathcal{S}$ who computes the $\mathcal{V}$ proof and sends to $\mathcal{Z}$. $\mathcal{S}$ also send accept messages from honest $\mathbb{V}$ to $\mathcal{Z}$ for C's operations. Like the previous aggregation step, $\mathcal{Z}$ could choose to create corrupt σ'_{agg} but this would fail verification when sent to $\mathcal{S}$ and the protocol would not proceed further. To proceed further, $\mathcal{Z}$ has to create a valid σ'_{agg} with the accept messages from $> |\mathbb{V}|/2$ members of $\mathbb{V}$. Finally $\mathcal{S}$ simulates Alice's signature over $(\sigma'_{\text{agg}}, M_A, M'_1, \dots, M'_n, pk_1, \dots, pk_n)$, creates tx_A and submits to $\mathcal{F}_{\text{bc}}$. The view of $\mathcal{Z}$, who controls C and $\mathbb{V}'$ will be $(l, \sigma_A, h, pk_A, p, s, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

5) Case 4: Alice is corrupt, C and all verifiers are honest

a) **Real-world**: Alice sends a $\mathcal{V}$ request to C and $\mathbb{V}$. A corrupt Alice could choose to create a corrupt σ_A but this would fail verification at the C and $\mathbb{V}$ and the protocol would stop. To proceed Alice has to compute valid (σ_A, h). C and $\mathbb{V}$ would proceed as normal and return a σ_{agg} to Alice. Alice could choose to send a corrupt $\mathcal{V}$ for verification to $\mathbb{V}$. Since verification would fail there would be no σ'_{agg} generated for Alice so she cannot proceed further. If Alice computes a valid $\mathcal{V}$ proof, C would return a σ'_{agg} to her and Alice eventually outputs tx_A. In tx_A Alice could choose to use a different M'_A but the hash of M'_A would not match the h signed in σ_{agg} and would fail verification in the smart contract which checks h matches M_A, and the l and h in σ'_{agg} are tied to M_A. Alice can only pass smart contract verification if she keeps the original M_A and valid M_{agg} and M'_{agg} in tx_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, \pi, y, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

b) **Ideal-world**: $\mathcal{S}$ needs to simulate the actions of $\mathbb{V}$ and C to $\mathcal{Z}$. $\mathcal{Z}$ picks a M_A and sends a request to C with hash h corresponding to M_A. $\mathcal{S}$ simulates C and $\mathbb{V}$ by assigning l to h and generating a σ_{agg}. σ_{agg} is sent to $\mathcal{Z}$ who computes the $\mathcal{V}$ proof. If $\mathcal{Z}$ decides to send a corrupted proof to $\mathcal{S}$, then it would fail verification and $\mathcal{S}$ would not generate a corresponding σ'_{agg}. The only way for $\mathcal{Z}$ to proceed is to compute valid $\mathcal{V}$ proof. Upon receiving valid proof $\mathcal{S}$ verifies it and generates σ'_{agg} which is sent to $\mathcal{Z}$. $\mathcal{Z}$ now creates tx_A and submits to $\mathcal{F}_{\text{bc}}$. The *code* associated with *SC.id* verifies σ'_{agg}, the hash of M_A included in tx_A matches $(h, l, \cdot)$ in σ_{agg}, and the l in previous tuple is same as in σ'_{agg}. The code also checks for freshness using the $block_{curr}$ value. If any of these checks fail verification then the smart contract would not execute in favor of $\mathcal{Z}$ and it would be identified as corrupt. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, p, s, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

6) Case 5: Alice and some verifiers are corrupt, C is honest

a) **Real-world**: As in Case 4, if Alice sends corrupt σ_A the C would fail verification and not proceed further. If the σ_A is valid, the C picks valid l and sends to all $\mathbb{V}$. The corrupt minority of $\mathbb{V}'$ could choose to not send their signatures or send corrupt signatures which the C can discard and generate a σ_{agg} from the honest majority in $\mathbb{V}$. Alice on receiving the σ_{agg} can choose to send an invalid $\mathcal{V}$ proof which would not generate accept signatures from the honest majority in $\mathbb{V}$. $\mathbb{V}'$ could choose to wrongly send accept signatures to C but since $|\mathbb{V}'| < |\mathbb{V}|/2$, C will not generate a σ'_{agg}. If Alice computed a valid $\mathcal{V}$ proof then she will receive a σ'_{agg} from C and Alice eventually outputs tx_A. As described in Case 4, Alice can only pass smart contract verification if she outputs a valid tx_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, \pi, y, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

b) **Ideal-world**: $\mathcal{S}$ needs to simulate the actions of $\mathbb{V} - \mathbb{V}'$ and C to $\mathcal{Z}$. $\mathcal{Z}$ picks a M_A and sends a request to C with hash h corresponding to M_A. If h or σ_A are invalid then $\mathcal{S}$ would not generate l and the protocol would stop. If valid request is received from $\mathcal{Z}$, $\mathcal{S}$ assigns l to h and sends to $\mathcal{Z}$. If $\mathbb{V}'$ controlled by $\mathcal{Z}$ send corrupt signatures to $\mathcal{S}$, it can just ignore those messages and output a σ_{agg} to $\mathcal{Z}$ by simulating the honest majority of $\mathbb{V}$'s actions. If $\mathcal{Z}$ decides to send corrupt proof to $\mathcal{S}$, then it would fail verification and $\mathcal{S}$ would not generate a corresponding σ'_{agg}. As in previous step, corrupt $\mathbb{V}'$ messages corresponding to $\mathcal{V}$ proof from $\mathcal{Z}$ can be ignored by $\mathcal{S}$. The only way for $\mathcal{Z}$ to proceed is to compute valid $\mathcal{V}$ proof. Upon receiving valid proof $\mathcal{S}$ verifies it and generates σ'_{agg} which is sent to $\mathcal{Z}$. $\mathcal{Z}$ now creates tx_A and submits to $\mathcal{F}_{\text{bc}}$. As described in Case 4, $\mathcal{Z}$ can only pass smart contract verification if tx_A contains valid signatures and M_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, p, s, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

7) Case 6: Alice, C are corrupt, all verifiers are honest

a) **Real-world**: If Alice sends corrupt σ_A to the C and $\mathbb{V}$, or if Alice sends a valid request but C chose a corrupt l similar to Case 2, the $\mathbb{V}$ will not send accept signatures to C since the request or l will not pass verification. The C can create corrupt σ_{agg} but this would fail verification eventually at the smart contract. If the σ_A is valid and the C picks valid l, the $\mathbb{V}$ will reply with accept messages so C can generate a valid σ_{agg}. Alice on receiving the σ_{agg} can choose to send an invalid $\mathcal{V}$ proof which would not generate accept signatures from the $\mathbb{V}$. Like the previous stage, the C can create corrupt σ'_{agg} but this would fail verification eventually at the smart contract. If Alice computed a valid $\mathcal{V}$ proof, the C would receive accept signatures from the $\mathbb{V}$ and C can compute a valid σ'_{agg}. Alice eventually outputs tx_A. As described in Case 4, Alice can only pass smart contract verification if she outputs a valid tx_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, \pi, y, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

b) **Ideal-world**: $\mathcal{S}$ needs to simulate the actions of $\mathbb{V}$ to $\mathcal{Z}$. $\mathcal{Z}$ picks a M_A and sends to $\mathcal{S}$, the hash h corresponding to M_A, σ_A, and l. If h or σ_A are invalid then $\mathcal{S}$ would not generate $\mathbb{V}$ signatures for $\mathcal{Z}$. $\mathcal{Z}$ can decide to proceed with the protocol by generating a corrupt σ_{agg} but this would fail verification in the *code* of the *SC.id* smart contract. If valid request and l is received from $\mathcal{Z}$, $\mathcal{S}$ sends $\mathbb{V}$ signatures to $\mathcal{Z}$ and $\mathcal{Z}$ can generate σ_{agg}. If $\mathcal{Z}$ decides to send corrupt proof to $\mathcal{S}$, then it would fail verification and $\mathcal{S}$ would not generate corresponding accept signatures from $\mathbb{V}$ to send to $\mathcal{Z}$. Like the previous stage, the $\mathcal{Z}$ can create corrupt σ'_{agg} but

this would fail verification eventually at the $SC.id$ smart contract. The only way for $\mathcal{Z}$ to proceed is to compute valid $\mathcal{V}$ proof. Upon receiving valid proof $\mathcal{S}$ verifies it and generates V signatures which are forwarded to $\mathcal{Z}$. $\mathcal{Z}$ generates a valid σ'_{agg}, creates tx_A, and submits to $\mathcal{F}_{\text{bc}}$. As described in Case 4, $\mathcal{Z}$ can only pass smart contract verification if tx_A contains valid signatures and M_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, p, s, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

8) Case 7: Alice, C and some verifiers are corrupt

a) **Real-world**: If Alice sends corrupt σ_A to the C and $\mathbb{V}$, or if Alice sends a valid request but C chose a corrupt l similar to Case 2, the honest majority $\mathbb{V} - \mathbb{V}'$ will not send accept signatures to C since the request or l will not pass verification. The C can create corrupt σ_{agg} but this would fail verification eventually at the smart contract. If the σ_A is valid and the C picks valid l, the honest majority $\mathbb{V} - \mathbb{V}'$ will reply with accept messages so C can generate a valid σ_{agg}. Alice on receiving the σ_{agg} can choose to send an invalid $\mathcal{V}$ proof which would not generate accept signatures from the honest majority $\mathbb{V} - \mathbb{V}'$. Like the previous stage, the C can create corrupt σ'_{agg} but this would fail verification eventually at the smart contract. If Alice computed a valid $\mathcal{V}$ proof, the C would receive accept signatures from the honest majority $\mathbb{V} - \mathbb{V}'$ and C can compute a valid σ'_{agg}. Alice eventually outputs tx_A. As described in Case 4, Alice can only pass smart contract verification if she outputs a valid tx_A. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, \pi, y, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

b) **Ideal-world**: $\mathcal{S}$ needs to simulate the actions of honest majority $\mathbb{V} - \mathbb{V}'$ to $\mathcal{Z}$. As in Case 6, $\mathcal{S}$ will not send any accept messages to $\mathcal{Z}$ when simulating the honest verifiers if $\mathcal{Z}$ sends corrupt σ_A and h on behalf of Alice and corrupt l on behalf of the C. $\mathcal{Z}$ will not be able to create valid σ_{agg} and σ'_{agg} since the honest majority of verifiers will be simulated by $\mathcal{S}$. The only way for $\mathcal{Z}$ to proceed is to submit valid h, σ_A and l to $\mathcal{S}$ and compute a valid σ_{agg} and $\mathcal{V}$ proof. Upon receiving valid proof $\mathcal{S}$ verifies it and generates $\mathbb{V} - \mathbb{V}'$ signatures which are forwarded to $\mathcal{Z}$. $\mathcal{Z}$ generates a valid σ'_{agg}, creates tx_A, and submits to $\mathcal{F}_{\text{bc}}$. As discussed in Case 4, $\mathcal{Z}$ can output a corrupt tx_A but the verification checks in the smart contract *code* will fail. $\mathcal{Z}$'s view will be $(l, \sigma_A, h, pk_A, p, s, tx_A, block_{curr}, pp, t_1, t_2, \alpha, \sigma_{V_i}, M_i, \sigma_{V'_i}, M'_i)$ for $i \in [1 \dots |\mathbb{V}|]$.

Let $\mathbb{TX}_{\mathcal{A}}$ represent the set of $\mathcal{A}$'s transactions, $\mathbb{TX}_{\mathcal{Z}}$ represent the set of $\mathcal{Z}$'s transactions, $\mathbb{TX}_{\mathcal{S}}$ represent the set of $\mathcal{S}$'s transactions, and let tx_A represent Alice's transaction in the pending pool, i.e., txpoolTable, respectively. Let the time elapsed since tx_A entered the txpoolTable be denoted by T_E, the maximum miner fee allowed per block by the blockchain system be blockMaxFee, let t_1 be the time taken for computing a given VDF, and let $\mathcal{P}(\mathbb{A})$ denote the power set of set $\mathbb{A}$. Then the advantage of $\mathcal{A}$ in winning the *FIRST* game against Alice, i.e., Alice's tx_A getting frontrunned is given by the following inequality:

$$\begin{aligned}\mathsf{Adv}_{\mathcal{A},FIRST}(\lambda) \leq Pr\Big[\Big(&\mathbb{TX}_B = min(|\mathbb{X}|);\\ &\mathbb{X} \in \mathcal{P}((\mathbb{TX}_{\mathcal{Z}} \cup \mathbb{TX}_{\mathcal{A}}) \cup (\mathbb{TX}_{\mathcal{S}} \setminus \{tx_a\}))\Big) \wedge\\ &\Big(\forall\, tx \in \mathbb{TX}_B,\ \sum tx.txfee \lessapprox \text{blockMaxFee}\Big)\\ &\wedge (T_E > t_1)\Big].\end{aligned}$$