Perils of Parallelism: Transaction Fee Mechanisms under Execution Uncertainty

Sarisht Wadhwa* *Duke University*
Aviv Yaish* *Yale University, IC3, Complexity Science Hub Vienna*
Fan Zhang *Yale University, IC3*
Kartik Nayak *Duke University*

Abstract

Modern blockchains increasingly rely on parallel execution to improve throughput. We show several industry and academic transaction fee mechanisms (TFMs) struggle to simultaneously account for execution parallelism while remaining performant and fair. First, if parallelism affects fees, adversarial protocol manipulations that offset possible benefits to throughput by introducing *fake* transactions become rational: users can insert functionally useless parallel transactions solely to reduce fees, and schedulers can create useless sequential transactions to increase revenue. Execution contingency, a core feature of expressive programming languages, both exacerbates the aforementioned threats and introduces new ones: (1) users may overpay for unused resources, and (2) scheduler revenue is harmed when reserved scheduling slots go unused due to contingency. We introduce a framework for this challenging setting, and prove an impossibility, highlighting an inherent tension: both parallelism and contingency involve a trade-off between minimizing risks for users and schedulers, as favoring one comes at the expense of the other. To complete the picture, we introduce fee mechanisms and prove that they achieve the boundaries of this trade-off. Our results provide rigorous foundations for evaluating designs advanced by notable blockchains, such as Sui and Monad.

1 Introduction

As the throughput of consensus protocols continues to improve, the throughput bottleneck of blockchains has switched to execution [27]. Parallel execution, i.e., running multiple transactions on parallel threads, is a prominent approach to increase execution throughput either used currently or slated for future upgrades by chains like Sui [11], Solana [10], Monad [26], and Ethereum [33]. However, in blockchains supporting Turing-complete smart contracts, resource contention among transactions cannot be decided without pre-execution. As a result, contention may undermine parallelism's benefits. E.g., if two *interdependent* transactions are scheduled to execute in parallel, one of the two would either have to fail and re-execute later (as in Monad [26]), or stall and lock system objects until it can resume executing. Transaction failure harms the system's user experience, while locking introduces overhead and forces serialization at contention points.

In both cases, execution is either slowed down or system's capacity is not fully used, defeating the purpose of parallelism. Worse, current designs do not charge transactions for locking capacity that eventually goes unused, thereby harming revenue [34]. A proper fee mechanism can alleviate these issues by pricing the externalities of contention and charging transactions that contribute significantly to the critical path. Thus, a natural question arises: *can we design a fee mechanism that accounts for the impact of transactions on parallel execution?*

To answer this, we must consider two "perils" that make the question hard: contingent transactions and shill transactions.

Peril 1: Contingent Transactions and Risk. Parallel execution amplifies the impact of *contingent transactions*: transactions which can access different object lists based on the state when executing. Such transactions create a tension between two forms of risk: *user risk*, where users overpay for objects that go unused, and *scheduler risk*, where execution capacity which is reserved ex-ante goes unpaid due to under-execution. We formalize both notions and prove a central impossibility result: *no mechanism can simultaneously eliminate both user risk and scheduler risk without either executing transactions in advance or collapsing to constant fees.*

Risk Allocation Mechanisms. Given this impossibility, we characterize a linear tradeoff between user and scheduler risk. We study three canonical mechanisms: (i) *user-friendly*, which charges only for realized execution, (ii) *scheduler-friendly*, which charges for worst-case declared execution, and (iii) *Even-Steven*, which splits unrealized cost evenly. We further analyze these mechanisms under three scheduler models: optimistic, pessimistic, and median, capturing different types of scheduler's utilities, when the execution of transactions is unknown. Together, these results provide a principled

*Equal contribution.

design space for fee mechanisms under execution uncertainty.

Peril 2: Shill attacks on Gas Consumption Mechanisms. We build on the Gas Computation Mechanism (GCM) introduced by prior work [2]. A GCM assigns an abstract notion of *effective compute* to each transaction given a schedule, while the TFM prices this compute based on user's priority. Unlike prior work, we place this framework in an explicitly adversarial setting: both users and the scheduler are modeled as rational, economically motivated parties who may inject strategically crafted *shill transactions*.

We identify two attack classes: *user shill attacks*, where fake transactions reduce the effective gas charged to a real transaction, and *scheduler shill attacks*, which inflate the gas charged to others. We formalize *shill-proofness* as a necessary security property for GCMs and show that it is fundamentally incompatible with the efficiency requirement that total gas equals the schedule makespan. This tension explains why several natural GCM proposals admit profitable manipulation, and motivates fee-based notions of shill resistance that account for both pricing and execution outcomes.

We use one *running example* throughout the paper. Alice sends a swap to an automated market maker that first reads an oracle price and only completes the trade (touching the pool and her balance) if the price is favorable; otherwise it reverts after the read. The set of objects it touches therefore depends on state it cannot see in advance, which makes it a contingent transaction. As a concrete user shill attack on this swap, suppose it touches a popular object and would normally be charged a high fee because it blocks other transactions from running in parallel. Alice can add a cheap, throwaway transaction that touches a *different*, idle object. The mechanism now sees more "parallel work" in the block and spreads the cost out, so Alice's real transaction is charged less, even though her extra transaction did nothing useful. A scheduler shill attack is the mirror image: the scheduler adds its own fake transaction to make honest transactions look like they consume more, so it can collect more in fees.

Shill Attacks with Contingent Transactions. If the shill transaction does not have to pay the complete price for each object it accesses (i.e., scheduler risk is non-zero), then the cost to perform shill attacks reduces. Thus transaction contingency influences the shill proofness of a protocol.

Object-Weighted Transaction Fee Mechanism. To translate gas consumption into payments under parallel execution, we introduce the *Object-Weighted Transaction Fee Mechanism (OW-TFM)*. Rather than charging purely based on serial execution time, OW-TFM prices a transaction according to the *objects it declares in the parallel schedule*, including both compute time and contention over shared objects. Intuitively, a transaction pays more not only when it runs longer, but also when it limits parallelism by using more objects. Formally, OW-TFM prices each object based on how often the object has been used in previous blocks, in a EIP-1559-like [28] price update function. OW-TFM then charges a fee equal to the sum of price of each object multiplied by the user-specified priority fee. We show parameters under which this price function leads to a shill-proof system.

Contributions. In summary, we contribute the following:

1. We show that transactions with conditional execution can cause either users to overpay for computation objects that are never used, or the scheduler to lose revenue due to contingent transactions underexecuting and thus underpaying. We establish an inherent trade-off in fee design: any attempt to reduce user risk (overpayment) increases scheduler risk (uncollected fees), and vice versa.

2. We introduce a formal framework to model parallel transaction execution with contingency for analyzing TFMs.

3. We identify user-level and scheduler level *shill attacks* where malicious party (user or scheduler) can add fake transactions to adversarialy decrease or increase fees.

4. We propose new transaction fee mechanism properties (*shill proofness*) and parameters spanning the user-scheduler risk spectrum, and present a variant of weighted area TFM that can satisfy these properties.

Outline. We begin by going over related work in Section 2. Then, we present our model in Section 3. In Section 4, we analyze how contingency in transactions can cause either overpayment or loss in revenue depending on whether a user pays even if its transaction under-executes. Section 5 maps the decision choices a designer must make when dealing with uncertain execution of transactions. Independently, we observe that previous works developing TFMs dependent parallel execution are vulnerable to shill attacks (Section 6). In Section 7, we see how contingent transactions make these attacks cheaper and easier to accomplish and define formal properties that every TFM design must satisfy. In Section 8, we look at object-weighted TFM (OW-TFM) which satisfies shill proofness. We conclude with a discussion of design choices and future directions that can be taken in Section 9.

2 Related Work

Parallel Execution and Blockchain Scalability. The transition from serial to parallel execution is well-documented in recent systems research. Ponnapalli et al. identified execution as the primary bottleneck in modern blockchains, leading to the development of parallel engines [27]. Prominent examples include Sui, which utilizes a fee-ordered DAG and object-based dependencies [11, 31], and Monad, which employs optimistic execution where conflicts trigger re-execution [26]. These systems rely on high-throughput consensus protocols like Mysticeti [4] and Bullshark [30] to order transactions rapidly

before execution. While these systems improve throughput, they introduce new overheads regarding resource locking and contention that standard serial fee markets do not address.

Transaction Fee Mechanisms (TFMs). Traditional TFMs, such as the first-price auctions in Bitcoin and the base-fee burning mechanism of Ethereum's EIP-1559, treat execution resources as fungible and one-dimensional [5, 7, 12, 14, 15, 17, 18, 29]. Recent literature has expanded this to *multidimensional fee markets*, where distinct resources (e.g., storage and compute) are priced separately to avoid congestion in specific domains. Diamandis et al. and Kiayias et al. explore the design of these multidimensional markets to better reflect resource scarcity [2, 8, 22]. Our work builds on this by incorporating the execution time and parallelism into the pricing model, a necessity for the safe operation of parallel chains.

Gas Computation in Parallel Settings. Our work directly builds on the framework proposed by Acilan et al., which introduced Gas Computation Mechanisms (GCMs) like the Shapley and Time-Proportional Makespan (TPM) approaches [2]. Their work established various properties which are good to have in GCM design, and propose various mechanisms to achieve (some of) the properties. We identify that these properties are insufficient to prevent *Shill Attacks*. We demonstrate that their proposed mechanisms allow adversaries to manipulate gas costs by injecting *fake transactions*, that have no added utility from its execution.

Scheduling and Complexity. The problem of block building has historically been modeled as a Knapsack Problem, optimizing fee revenue under a gas limit [1, 24]. In parallel environments, this transforms into a scheduling problem where transaction order dictates concurrency. We analyze greedy heuristics used by systems like Sui, and random heuristics like Aptos and Monad, showing they are revenue-suboptimal compared to optimal parallel schedules. Furthermore, we ground our impossibility results in complexity theory, specifically the limits of parallel computation. We leverage standard results regarding the P-completeness of generic machine simulation to prove the hardness of verifying contingent object usage.

3 Model

Transactions. A transaction $tx(t, g, \text{ins}, \{R, W\})$ is a (finite) sequence of instructions that interact with the blockchain state. The parameter t denotes the (upper bound on) compute units required to execute the transaction on a *standard machine*.[1] Parameter g denotes the price the user is willing to pay for all compute used. The instruction sequence (ins) determines the control flow (conditionals, reads, writes) and how objects in the read and write sets (R, W) are accessed. R is the set of objects that the transactions accesses, but does not modify, whereas W represents the set of objects that it modifies. Since metadata beyond compute time and object access is often immaterial for scheduling, we represent transactions as $tx \simeq (t, g, \{R, W\})$. A transaction's declared read and write sets may be conservative: a transaction may declare *contingent* objects that are only conditionally accessed at runtime.

Schedule. A *schedule* $\mathcal{S}$ is an ordered set of transactions that are to be executed in the given block, in which any parent of a transaction is executed strictly before the child. Typically, a schedule is limited by the amount of gas that all transactions in the schedule combined can consume. However, in this paper, we would constrain the schedule by the total compute consumed by all transactions (i.e., ignore other aspects of gas computation such as storage). The schedule is generated from a publicly known algorithm, which we label as the *scheduler*.

The revenue of the scheduler is the total *fee* received from all the scheduled transactions. We represent the predicate that a transaction is in the schedule by $tx \in \mathcal{S}$. We use the term prefix schedule to define a schedule until a particular transaction, i.e., $\mathcal{S}_{tx}$, which represents the schedule with all transactions that must be executed before the transaction tx.

State and Execution. Let s denote the global blockchain state (collection of object values) at a point in time. Given a schedule $\mathcal{S}$, let $s \xrightarrow{\mathcal{S}} s'$ denote the execution of the transactions in $\mathcal{S}$ (in order) on state s, producing state s'. The *execution outcome* of running a transaction tx on state s is written as $\text{Exec}(s, tx) = \text{out}$, where out is the result of executing tx.

Objects. An object o can be thought of as data that each transaction reads, modifies, or creates on the blockchain. The set of all objects is represented as O, and $n_O = |O|$. Each transaction $tx(t, g, \{R, W\})$ interacts with the objects in the sets R and W. For all objects in R, the object remains unchanged after the interaction. For all objects in W, the object can either be created or, if it already exists, then the value is modified. We represent the value of the object o by $\text{val}(o)$. Sometimes a transaction might not access all the objects it asks for.

Objects have been used before by various blockchains, e.g., SUI Packages are smart contracts and functions (a type of read-only objects) while SUI Objects represent storage that each transaction can access [31].

We introduce a single *running example* that we reuse throughout the paper to make each new idea concrete.

Example 1 (Running Example: Alice's Swap)**.** *Alice submits a transaction tx_{A} to an automated market maker (AMM) [6] which first* reads an oracle price *object o_{px}. If the price is favorable, it executes the swap, writing the pool reserves o_{pool} and Alice's balance o_{bal}; if the price is unfavorable, it reverts right after reading the oracle and touches nothing else. So tx_{A} declares the objects $\{o_{\mathsf{px}}, o_{\mathsf{pool}}, o_{\mathsf{bal}}\}$, but only o_{px} is* always *used; o_{pool} and o_{bal} are* contingent*, used only on the favorable branch. Throughout, we assume tx_{A} uses one unit of compute, has priority* 1*, and that o_{pool} is used thrice*

[1] Standard machine specifications, e.g., covering CPU and memory requirements, provide a fixed reference profile to measure t.

as much as other objects. We call the favorable branch the desired *execution, and the revert the* under-executed *one. For this, we define contingency and some related terms below.*

Used Sets. As a transaction's declared read/write sets may be conditional, we define the *used* sets of tx when executed on a particular state (or after a prefix schedule) as $X_R(s,tx) \subseteq R(tx)$ and $X_W(s,tx) \subseteq W(tx)$, i.e., the subset of declared objects actually accessed during execution. When tx is run in the context of a schedule $\mathcal{S}$ applied to initial state s, the used sets are dependent on the prefix $\mathcal{S}_{tx}$ and we thus write $X_R(s,\mathcal{S}_{tx},tx)$ and $X_W(s,\mathcal{S}_{tx},tx)$ when clarity is required.

Contingent Objects. Intuitively, an object is contingent for tx if depending on prior transactions, the transaction conditionally accesses the object.

Definition 1 (Contingent Object)**.** *A declared object $o \in R(tx) \cup W(tx)$ is a* contingent object *for tx from the perspective of some initial state s if there exist two legal prefix schedules drawn from the possible transaction sets $\mathcal{S}_{tx}$ and $\mathcal{S}'_{tx}$ such that $s \xrightarrow{\mathcal{S}_{tx}} s'$, $s \xrightarrow{\mathcal{S}'_{tx}} s''$, where either the read used set differs (i.e., $o \in X_R(s,\mathcal{S}_{tx},tx)$ but $o \notin X_R(s,\mathcal{S}'_{tx},tx)$), or the write used set differs (meaning: $o \in X_W(s,\mathcal{S}_{tx},tx)$ but $o \notin X_W(s,\mathcal{S}'_{tx},tx)$).*

User Declarations. We consider the possibility of users declaring contigent object sets in their transactions, and denote these sets by $C_R(tx) \subseteq R(tx)$ and $C_W(tx) \subseteq W(tx)$.

Contingent Transactions. A transaction is contingent iff at least one of its declared objects is a contingent object, which can be formalized as follows.

Definition 2 (Contingent Transaction)**.** *A transaction tx is a* contingent transaction *iff there exist two prefix schedules $\mathcal{S}_{tx}$ and $\mathcal{S}'_{tx}$ such that $s \xrightarrow{\mathcal{S}_{tx}} s', s \xrightarrow{\mathcal{S}'_{tx}} s''$, and the objects accessed during tx's execution on these two outcome states differ:*

$$X_R(s',tx) \neq X_R(s'',tx) \ \vee \ X_W(s',tx) \neq X_W(s'',tx)$$

Of these outcomes, one corresponds to the user's desired execution. This outcome is assumed to use the contingent objects. The other *under-executed* outcome do not use the contingent objects and potentially uses less compute. Note that contingency is about *which* objects a transaction ends up touching, not *what values* those objects hold. A transaction that always reads and writes the same objects is *not* contingent, even if an earlier transaction changed the values stored in those objects (e.g., copying a variable a modified by a prior transaction). It becomes contingent only when earlier transactions can change the set of objects it actually accesses.

Example 2. *We now provide several examples of contingency.*

- *A transaction that conditionally reads an oracle price and only writes balances if the price exceeds a threshold is contingent: the oracle object is a contingent object because prior transactions may have changed its value.*
- *A transaction that submits a swap to router which redirects the swap to the AMM that offers the best price is contingent. All AMMs' liquidity are contingent objects. However, in this transaction, the scheduler should expect the transaction not to access all objects. Such examples are out of scope and left as future work (see Section 9).*
- *A pure transfer that reads only its own nonce and always writes the same set is non-contingent if no declared object can vary the control flow.*

Transaction Conflicts. As in prior works, transactions tx_1 and tx_2 are said to *conflict* if they access a common object and at least one access is a write. Formally,

$$C(tx_1,tx_2) \iff \{(R(tx_1) \cap W(tx_2)) \cup (W(tx_1) \cap R(tx_2)) \cup (W(tx_1) \cap W(tx_2))\} \neq \emptyset.$$

If $C(tx_1,tx_2)$ then tx_1 and tx_2 cannot be executed in parallel. Contingency interacts with conflicts: if an object is contingent and ends up unused in a given execution, the conflict may not materialize for that run; thus the conflict relation may be *execution-dependent* when contingent objects are present.

Fees and Effective Compute. Each transaction specifies a *maximum fee* and a *compute price* (price per unit of effective compute[2]). While traditional serial blockchains equate effective compute with execution time (t), parallel execution introduces a dependency: a transaction's cost should depend on its ability to run concurrently with the rest of the schedule.

Modeling Contingency. Contingent transactions introduce uncertainty: declared objects may not be used at runtime. To analyze the gap between declared and realized execution, we define the following definitions for fees for a transaction tx executed on state s under schedule $\mathcal{S}$. To cover fee burning, we also distinguish between the *fee paid* by the user (f) and the *fee received* by the scheduler (r). We use f and r as the generic paid and received fees; subscripts pick out specific reference points. In particular, f_{att}, f_{base}, and f_{ui} (defined below) are hypothetical fees used for comparison, while f_{act} and r_{act} are the *actual* (realized) amounts paid by the user and received by the scheduler for a given execution.

- **Attainable Fee (f_{att}):** The fee tx would pay if it declared all contingent objects as definitively used. This represents the maximum theoretical cost (no uncertainty).
- **Baseline Fee (f_{base}):** The fee tx would pay if it declared only its deterministic (guaranteed used) objects. This represents the minimum guaranteed cost.
- **User-Ideal Fee (f_{ui}):** The fee corresponding to exactly the objects tx actually uses during execution (U_{used}).

[2]Effective compute is equivalent to gas used, however, only the compute part of gas consumption is useful for us and thus in accordance to the multi-dimensional fee proposals, any other fee is separated.

- **Actual Paid Fee (f_{act}) and Received Fee (r_{act}):** The fee actually charged to the user and the portion received by the scheduler after burning, respectively.

We impose two monotonicity constraints on these fees:

1. $f_{\text{base}} \leq f_{\text{act}}$: Users cannot reduce fees below the baseline cost of their guaranteed execution path.

2. $f_{\text{act}} \leq f_{\text{att}}$: Declaring uncertainty (contingency) should not cost more than declaring certainty.

Example 3 (Running Example: The Four Fees for Alice's Swap)**.** *Recall tx_{A} declares $\{o_{\mathsf{px}}, o_{\mathsf{pool}}, o_{\mathsf{bal}}\}$, say with object costs $1,3,1$. The attainable fee treats all declared objects as used, so $f_{\text{att}} = 1+3+1 = 5$. The baseline fee counts only the always-used oracle, so $f_{\text{base}} = 1$. The user-ideal fee depends on the branch taken: in the desired (favorable) execution Alice touches everything, so $f_{\text{ui}} = 5$; in the under-executed (revert) branch only the oracle is touched, so $f_{\text{ui}} = 1$. The interesting case is the under-executed one, where $f_{\text{base}} = f_{\text{ui}} = 1$ sit far below $f_{\text{att}} = 5$: the question of who pays for the 4 units of unused, but reserved, capacity is discussed in Section 4.*

Scheduler Incentives and Burning. In mechanisms like EIP-1559 [28], some of the paid fee can be burned. We denote the *retention ratio* as $\gamma = r/f$. To prevent schedulers from manipulating execution outcomes (e.g., forcing transactions to fail or under-execute to minimize work while collecting fees), we introduce a dynamic burning constraint: the retention ratio must increase as the transaction utilizes more of its declared objects. For simplicity, we assume that γ is constant. This assumption aligns with practice. For example, EIP-1559 defines for each block some fixed base fee which is burnt from each included transaction.

3.1 Adversarial Model

We model all parties as rational and economically motivated. There are two strategic roles: (i) *users*, who submit transactions, and (ii) the *scheduler*, which selects an execution schedule $\mathcal{S}$. Consensus is assumed honest and only determines the candidate transaction set $\mathcal{T}$; however a user or scheduler can insert arbitrary transactions in $\mathcal{T}$.

We assume that all parties know the transaction fee mechanism (TFM), gas computation mechanism (GCM), and the risk-division rule that would be defined in Section 5. They can form beliefs about the outcome of contingent executions but cannot predict on-chain execution of transactions.

User Actions. A user may: (a) mis-declare contingent objects to influence f_{ui} or f_{att}; and (b) submit *shill transactions* which lower the total fee paid. As a user's valuation $\text{val}(tx)$ is private, each one acts to maximize its expected utility $U_{\text{user}} = \text{val}(tx) - f_{\text{act}}$ by minimizing f_{act}.

Scheduler Actions. The scheduler may: (a) insert shills if profitable, or (b) schedule transactions with favorable contingency risk. Its utility equals the total collected fees: $U_{\text{sched}} = \sum_{tx \in \mathcal{S}} (r_{\text{act}}(s, \mathcal{S}, tx))$. Since r_{act} is a constant fraction of f_{act}, we instead consider the scheduler's objective in terms of f_{act}. Thus, the scheduler's goal is to maximize the *total* f_{act} summed over all transactions it includes in the schedule, rather than the fee of any single transaction in isolation.

4 Contingent Transactions

We now turn our focus to *contingent transactions*, that is, transactions that do not execute as expected.

4.1 User Risk and Scheduler Risk Tradeoff

In what follows, we prove that contingency poses a risk to users and schedulers: unless the scheduler has exponential compute, a protocol can never mitigate both types of risk at the same time. To quantify the tradeoff, we define these risks.

The user risk due to contingency entails paying for unused declared objects, as we now formalize.

Definition 3 (User Risk)**.** *For a transaction tx executed in schedule $\mathcal{S}$ from initial state s, the* user risk *is*

$$\text{UR}(s, \mathcal{S}, tx) \;:=\; f_{\text{act}}(s, \mathcal{S}, tx) - f_{\text{ui}}(s, \mathcal{S}, tx). \tag{1}$$

That is, the user risk is the amount the user actually pays in excess of the user-ideal fee.

However, if contingent objects are not paid for, then contingent transactions do not pay their attainable value. This reduces the scheduler's utility and thus also increases its risk. As a reminder, paid fee is a proxy for the scheduler's utility, and this extends to the definition of risk.

Definition 4 (Scheduler Risk)**.** *For a transaction tx scheduled under $\mathcal{S}$ from initial state s, the* scheduler risk *is*

$$\text{SR}(s, \mathcal{S}, tx) \;:=\; f_{\text{att}}(s, \mathcal{S}, tx) - f_{\text{act}}(s, \mathcal{S}, tx). \tag{2}$$

Thus the scheduler risk measures the shortfall between the fee the scheduler expected in the best configuration and the fee actually collected.

Our two risk metrics show the difference between protocols that make a user pay for all declared objects and those that require payment only for used objects.

Lemma 1 (Risk Tradeoffs)**.** *For any transaction tx, initial state s and schedule $\mathcal{S}$, then the sum of* UR *and* SR *is constant, independent of the execution of the transaction.*

Proof. By definition, $\text{UR} + \text{SR} = f_{\text{act}} - f_{\text{ui}} + f_{\text{att}} - f_{\text{act}}$, so:

$$\text{UR}(s, \mathcal{S}, tx) + \text{SR}(s, \mathcal{S}, tx) \;=\; f_{\text{att}}(s, \mathcal{S}, tx) - f_{\text{ui}}(s, \mathcal{S}, tx).$$

□

By Lemma 1, the sum of user and scheduler risks is constant. Thus, any protocol that reduces user risk must increase scheduler risk for that transaction (and vice-versa), unless the attainable fee is the same as the user ideal fee ($f_{\text{att}} = f_{\text{ui}}$).

We note one subtlety: even though UR + SR is fixed, the *combined* utility of the user and the scheduler is not. The reason is burning. Because the scheduler only keeps a fraction γ of what the user pays (the rest is burned), money shifted from the user to the scheduler is not a clean one-to-one transfer. We make this dependence on γ explicit when we analyze the mechanisms in Sections 5 and 7.

Example 4 (Running Example: Splitting the Risk)**.** *Take Alice's swap on its under-executed branch, where $f_{\text{att}} = 5$ and $f_{\text{ui}} = 1$, so the total risk to divide is $f_{\text{att}} - f_{\text{ui}} = 4$, regardless of how we price it. Writing $f_{\text{act}} = \alpha f_{\text{att}} + (1-\alpha) f_{\text{ui}}$ for a knob $\alpha \in [0,1]$ (detailed in Section 5), the split is: at $\alpha = 0$ Alice pays $f_{\text{act}} = 1$, so $\text{UR} = 0$ and $\text{SR} = 4$ (the scheduler eats the wasted capacity); at $\alpha = 1$ Alice pays $f_{\text{act}} = 5$ for objects she never touched, so $\text{UR} = 4$ and $\text{SR} = 0$; at $\alpha = \frac{1}{2}$ she pays $f_{\text{act}} = 3$, splitting the risk evenly ($\text{UR} = \text{SR} = 2$). In every case $\text{UR} + \text{SR} = 4$: the trade-off only decides* who *bears the 4 units, never makes them disappear.*

Intuitively, if the scheduler cannot determine the execution outcome (the used sets and observable outputs) of transactions when it forms a schedule, then it cannot simultaneously set the f_{act} to be close to the user ideal fee (f_{ui}) and the maximum fee it can yield (f_{att}). We formalize this intuition.

Lemma 2. *If the scheduler does not execute transactions while constructing the schedule, then in the general case it cannot decide whether some contingent transaction tx will access a contingent object o in its execution.*

The proof follows from the fact that if the schedule has not been executed, then the state s' at which the contingent transaction is executed cannot be distinguished from whether it uses its contingent object or not. Formally,

Proof. Let s' and s'' be states such that $\text{Exec}(s', tx) \neq \text{Exec}(s'', tx)$ (by the definition of contingent transactions, such states much exist). Since the state before execution is not determined, it is unknown which execution route the transaction would take, and thus it is unknown if object o is accessed. □

Theorem 1. *Let G denote the block compute limit used to validate schedule feasibility. If a scheduler is constrained to decide schedules with at most G compute available (i.e., cannot perform more compute than the schedule itself), then for a schedule containing at least two conflicting contingent transactions, it is impossible to simultaneously set payments so that $\text{UR}(\cdot) = 0$ and $\text{SR}(\cdot) = 0$ for all possible transaction and state inputs, except in a constant fee structure where the fee paid is independent of the objects used.*

Proof. Suppose that the scheduler generates a schedule $\mathcal{S}$ guaranteeing $\text{UR}(s, \mathcal{S}, tx) = 0$ and $\text{SR}(s, \mathcal{S}, tx) = 0$ for every tx, s. By Lemma 1, we would have for every tx that $f_{\text{act}}(s, \mathcal{S}, tx) = f_{\text{ui}}(tx)$ and $f_{\text{act}}(s, \mathcal{S}, tx) = f_{\text{att}}(tx)$. Therefore, $f_{\text{att}}(tx) = f_{\text{ui}}(tx)$ for every tx. This implies that a transaction declaring less objects (f_{ui}) pays the same fee as when declaring a super-set of the objects (f_{att}). Therefore, the fee paid is independent of the objects it accesses.

To ensure both risks are zero, the scheduler needs to know which contingent objects will be used (to charge exactly the user ideal fee), and adjust its schedule accordingly. By Lemma 2, this entails executing transactions.

Thus, the scheduler has a G computation available to execute transactions and its goal is to compute a schedule within G computation limit. To do so, the scheduler must at least execute each transaction in the schedule, which implies a computation greater than G. □

Remark 1. *Theorem 1 shows that when faced with conflicting contingent transactions, simultaneously eliminating user and scheduler risk is only possible in three cases. The first is the "trivial" solution: Only include non-conflicting transactions and thus avoid any contingency. The second is the constant fee structure, i.e., all transactions pay the same fee. Lastly, we can burn all the revenue that the scheduler makes, while requiring the user to pay only for the objects it uses.*

4.2 Computational Impossibility

Our impossibility is an *economic* statement: under the budget G, the scheduler cannot zero out both risks at once. The natural follow-up question is *why* the scheduler cannot spend some extra effort to figure out which contingent objects get used and price them exactly. The rest of this section answers that with a *computational* argument: deciding object usage is essentially as hard as running the transaction itself, so there is no cheap shortcut. This is what makes the trade-off fundamental rather than an artifact of a weak scheduler, since no scheduler can predict usage without execution.

Object Usage Machine and Complexity. To make the hardness statement precise, we prove that there are no "short-cuts" to verifying contingency in general case. Put differently, asserting whether an object declared by a transaction is used is sequential, when considering a general transaction specified in a Turing-complete language. We proceed with the required technicalities. Definition 5 formalizes an abstract machine that determines whether a transaction uses a given object.

Definition 5 (Object Usage Machine (OUM))**.** *Given transaction tx, an initial state s, an object o, and a gas limit G, the* Object Usage Machine *executes tx on s, and, if o is accessed before G gas is used, then the machine halts and outputs ACCEPT. Otherwise, the machine halts when the gas budget is exhausted and outputs REJECT.*

We define a natural decision problem using Definition 5.

Definition 6 (Object Usage Decision Problem (OUDP))**.** *Given (o, tx, s, G) where the gas limit G is specified in unary, does the OUM output ACCEPT?*

If a method to *efficiently* decide Definition 6 on any general input exists, then it could be utilized by schedulers to quickly verify contingency. To understand what efficiency means in our case, note that for the performant protocols considered, block-time is strictly lower than the highest possible block execution time given the protocol's per-block gas limit and its reference hardware specifications. Thus, the minimal efficiency requirement entails *at least* verifying contingency in sub-polynomial time with respect to gas.

However, we show in Theorem 2 that OUDP is ***P***-complete. Under widely believed complexity class separations [19], this implies that deciding a general transaction's object usage at execution step t requires sequential work proportional to t. In turn, this means the user-scheduler risk trade-off is unavoidable if transactions can execute Turing-complete code.

Theorem 2 (***P***-completeness of OUDP)**.** *The Object Usage Decision Problem (OUDP) is* **P***-complete.*

Proof. To show completeness, we need to prove that OUDP is in ***P*** and that it is ***P***-hard. First, recall the definition of ***P***.

Definition 7 (Class ***P***)**.** *The class* **P** *equals the set of all decision problems (i.e., that produce a single output bit for any input) solvable by a deterministic Turing machine in polynomial time in input length.*

Now, we can prove that indeed OUDP is in ***P***. Given an instance (o, tx, s, G) where the gas limit G is specified in unary, then one can simulate the execution of tx on the OUM step-by-step until reaching the limit. This takes polynomial time in the input size and thus that OUDP is in ***P***, as G is encoded in unary. Note that unary encoding is the literature's standard approach for such reductions [19]. Intuitively, that is to ensure that the running time permitted by G is polynomial in input length. To see why, consider an input (o, tx, s, G). As G is a sub-string of this input, then the input's length is greater than or equal to the length of G. Given a binary encoding, then even the simplest step-by-step execution for G steps would formally require $O\left(2^G\right)$ time, exponential in input length.

We continue by showing that OUDP is ***P***-hard. To do so, we reduce from the Generic Machine Simulation Problem (given in Definition 8), well-known to be ***P***-complete [19].

Definition 8 (Generic Machine Simulation Problem (GMSP))**.** *Given a description $\bar{T}$ of a Turing machine T, a unary-encoded integer n, and an input string x, does T accept x within n steps?*

Given a GMSP instance $(\bar{T}, n, x)$, our reduction proceeds by constructing an OUDP instance (o, tx, s, G), thus tying OUDP and GMSP together. Intuitively, we nest a universal Turing machine to simulate T on top of the blockchain's VM.

A classic complexity theory result shows that there exists a universal Turing machine that can simulate the execution of any arbitrary machine [19]. Thus, set tx to equal the universal machine's implementation in the blockchain's VM that performs n steps, and which, upon reaching an accepting state, accesses a fixed object o not involved in the implementation. Note that the length of tx is constant and independent of the GMSP instance. Next, we set the state s to equal the literature's canonical representation of GMSP instances as used for reductions, defined as follows. Let l be the input's length, # be a delimiter character which cannot be present in input strings, $p(l)$ be an upper bound on the running time of T such that $p(l) = 2^{k_1 \lceil \log(l+1) \rceil}$ where k_1 is some integer power of two (traditionally chosen so that the encoding equals a turned-on bit followed by a trail of zeros, and thus is computable by bit-shifts [19]), and define $\#^{p(l)}$ as the delimiter concatenated to itself $p(l)$ times. Then, the canonical representation is $x\#\bar{T}\#^{p(l)}$. Finally, given that g_1 is the maximal gas consumed to execute one step of the machine T using the fixed machine encoded in tx and that g_2 is the gas needed to access o, set G to $2^{k_2 \lceil \log((g_1 \cdot n) + g_2) \rceil}$ for some integer power of two k_2.

In total, this is an efficient reduction such that tx accesses o within the limit G if and only if T accepts x within n steps. Combined with the ***P***-completeness of GMSP, we get that OUDP is ***P***-hard. In turn, as OUDP is both in ***P*** and ***P***-hard, we conclude it is ***P***-complete. □

Remark 2. *Our results show that the risk tradeoff is ex ante: the scheduler must reserve capacity and price uncertainty* before *execution. This issue cannot be mitigated using ex post optimizations like pruning unused contingent objects, which require execution. Indeed, Ethereum's recent block-level access lists proposal makes a similar distinction [33]. Thus, while the proposal requires recording touched objects when executing transactions at block creation time (to make later verification amenable to parallelization), it still emphasizes the possibility of adversarial "phantom" over-declarations.*

5 Mapping the Scheduler Design Space

Given the fundamental impossibility of simultaneously minimizing both user risk (UR) and scheduler risk (SR) when dealing with contingent transactions, we explore three fee division mechanisms that allocate this risk differently between the two parties. Two of these correspond to the theoretical boundaries, that is, assign all risk to either users or schedulers, while the third balances risk equally.

5.1 Risk Division

We present the *user-friendly*, *scheduler-friendly*, and *Even-Steven* risk division mechanisms. Each defines how paid fees

(f_{act}) relate to attainable (f_{att}) and user-ideal (f_{ui}) fees.

5.1.1 User-Friendly Mechanism

Under the *user-friendly* fee division mechanism, users pay for exactly what they use, thus eliminating user risk entirely. That is if the contingent transaction under-executes, then the user pays a fee that a transaction expecting this execution would have paid. Formally, the paid fee equals the user-ideal fee: $f_{\text{act}} = f_{\text{ui}}$. This implies that the scheduler absorbs all uncertainty associated with contingent execution, leading to

$$\text{SR} = f_{\text{att}} - f_{\text{act}} = f_{\text{att}} - f_{\text{ui}}, \quad \text{and} \quad \text{UR} = 0.$$

Intuitively, this is a user-oriented design: one only pays for the part of the transaction that actually executes. But, the scheduler bears all risk, as its realized revenue decreases whenever transactions under-execute. Using our formal notations, if the transaction executes completely, then $f_{\text{ui}} = f_{\text{att}}$, and the scheduler receives the maximum possible fee. When a transaction under-executes, however, f_{ui} may decrease to the baseline fee f_{base}. Such a design is commonly seen in Ethereum-esque fee mechanisms, where parallelism is not a factor.

5.1.2 Scheduler-Friendly Mechanism

The *scheduler-friendly* fee division lies at the opposite extreme: the scheduler bears no risk and receives the attainable fee regardless of execution outcomes: $f_{\text{act}} = f_{\text{att}}$. This implies $\text{SR} = 0$ and $\text{UR} = f_{\text{act}} - f_{\text{ui}} = f_{\text{att}} - f_{\text{ui}}$. Under this mechanism, the user must pay the full attainable fee *even* if part of its transaction does not execute as intended. The scheduler's revenue is thus insulated from contingency-related risks. Such a design is proposed for chains like Monad [25].

Scheduler-friendliness can be interpreted as an *execution-agnostic* pricing rule: users always pay the highest possible fee and are not reimbursed post-execution for unused objects. This guarantees revenue certainty for the scheduler, but exposes users to potential overpayment when contingent transactions do not make use of all declared objects.

5.1.3 The Even-Steven Mechanism

Between these two extremes lies the *Even-Steven* mechanism, which evenly splits the risk between the user and the scheduler. Under our notation, this can be formalized as requiring: $f_{\text{act}} - f_{\text{ui}} = f_{\text{att}} - f_{\text{act}}$. Thus, solving for the paid fee yields:

$$f_{\text{act}} = \frac{f_{\text{att}} + f_{\text{ui}}}{2}. \tag{3}$$

This equal-risk division ensures that the user and scheduler each bear half the uncertainty implied by contingency.

The Even-Steven mechanism can be viewed as a compromise: the user pays for half of the unrealized execution of the transaction. If a transaction completely fails to execute, the user still pays halfway between the attainable fee and the user-ideal fee. Conversely, if the transaction executes successfully, both user and scheduler pay the optimal amount.

5.1.4 Generalized Representation

The three mechanisms can be represented as points on a linear spectrum parameterized by $\alpha \in [0,1]$: $f_{\text{act}} = \alpha f_{\text{att}} + (1-\alpha) f_{\text{ui}}$. Here, α encodes how much risk is "transferred" to the user:

- $\alpha = 0$: **User-Friendly** (no user risk),
- $\alpha = \frac{1}{2}$: **Even-Steven** (equal risk),
- $\alpha = 1$: **Scheduler-Friendly** (no scheduler risk).

5.2 Revenue Maximization Functions

The aim for each scheduler is to maximize the utility generated given contingent transactions. For this, a scheduler can choose among utility functions that optimize different aspects of revenue, each operating based on a different expectation regarding the successful execution of contingent transactions.

The first scheduler maximizes revenue under the assumption of the most favorable possible execution outcome, utilizing the highest potential fee achievable.

Definition 9 (Optimistic Scheduler)**.** *The* Optimistic Scheduler *optimizes the best possible revenue under the assumption that* all transactions successfully execute.

In the context of fees, if a transaction executes successfully and uses all declared objects, the user-ideal fee paid approaches the attainable fee ($f_{\text{ui}} = f_{\text{att}}$). The optimistic scheduler maximizes revenue based on the expectation that the paid fee (f_{act}) equals the attainable fee (f_{att}) for all scheduled transactions, thus optimizing based on the upper bound of potential income. As scheduler expects $f_{\text{ui}} = f_{\text{att}}$, all risk divisions lead to the same utility, $f^s_{\text{act}} = f_{\text{att}}$.

The second scheduler minimizes risk exposure by planning for the worst-case revenue scenario.

Definition 10 (Pessimistic Scheduler)**.** *The* Pessimistic Scheduler *assumes the worst possible execution for all contingent transactions and* optimizes the worst possible revenue.

A user would pay f_{base} for submitting a deterministic, non-contingent transaction with the same minimum computational and always-used object requirements. This fee f_{base} is a lower bound, since $f_{\text{base}} \leq f_{\text{ui}}$. Thus, for a scheduler expecting the worst possible fee from the transaction $f^s_{\text{act}} = f_{\text{base}}$.

From eq. (3), the worst-case fee paid under the Even-Steven mechanism is $f_{\text{act}} = \frac{f_{\text{att}} + f_{\text{base}}}{2}$. The pessimistic scheduler therefore optimizes revenue based on this minimum guaranteed fee that would be received even the transaction under-executes.

The third scheduler utilizes a probabilistic assumption to balance optimism and pessimism regarding contingency.

Definition 11 (Median Scheduler)**.** *The* Median Scheduler *is defined as one that assumes that half the contingent transactions will under-execute. This implies that,* in expectation, it receives halfway between the potential revenue and minimum revenue *from the contingent transaction.*

Given the assumptions of this scheduler, it would expect the paid fee to equal $f_{\text{act}} = \frac{3f_{\text{att}}+f_{\text{base}}}{4}$.

We provide a summary of the different combinations of revenue maximization functions and risk divisions in Table 1.

Table 1: Scheduler utility under different risk divisions and execution outcome expectations.

	Optimistic	Pessimistic	Median
User-Friendly	f_{att}	f_{base}	$\frac{f_{\text{att}}+f_{\text{base}}}{2}$
Scheduler-Friendly	f_{att}	f_{att}	f_{att}
Even-Steven	f_{att}	$\frac{f_{\text{att}}+f_{\text{base}}}{2}$	$\frac{3f_{\text{att}}+f_{\text{base}}}{4}$

6 Shill Attacks on GCMs

We now analyze shill attacks on the GCMs presented in the literature and introduce properties required for a mechanism to be robust to these manipulations. In such attacks, *fake* transactions are introduced, either by the scheduler to increase the fee paid by users, or by users to reduce their expenses.

The specific attacks that we analyze target parallel-execution blockchains. Previous works like [2, 8, 22, 23] note accurately pricing transactions requires considering that while two transactions' computation time might be the same, one may access more *in-demand* objects than the other. In particular, Acilan et al. [2] describe a set of important properties and suggest various GCM designs that satisfy subsets of these properties. We believe one property that was not considered, but should be of the highest importance, is shill-proofness.

As GCMs were first put forth by Acilan et al. [2], we follow their assumption that the fee paid per gas unit is the same for all transactions until otherwise specified, and further assume in this section that user payments are fully received by the scheduler. Shills are priced exactly like any honest transaction and must pay whatever the mechanism at hand prescribes, including any burned portion. An attack is therefore only worthwhile when the gas (or fee) saved on the target transaction outweighs what the adversary must pay for shills, and such considerations are fully accounted for in our analysis.

6.1 Attacking Existing GCMs

We begin by demonstrating how shill attacks can be applied to two of the GCMs proposed by [2]. Let $\mathcal{T}$ be the set of transactions scheduled, $t(tx)$ the execution time of tx, O the set of objects accessed (analogous to the notion of storage keys considered by [2]), and $v(S)$ the least amount of time taken to execute a subset $S \subseteq \mathcal{T}$.

Definition 12 (Shapley GCM)**.** *Allocates gas used via Shapley value (marginal contributions to the makespan averaged over all subsets), that is,*

$$\mathit{gas}_{\mathcal{T}}^{\mathit{Shapley}}(tx) = \sum_{S \subseteq \mathcal{T}\setminus\{tx\}} \frac{v(S\cup\{tx\}) - v(S)}{|\mathcal{T}|\binom{|\mathcal{T}|-1}{|S|}}.$$

Definition 13 (Time-Proportional Makespan (TPM) GCM)**.** *Assigns gas used proportional to the share of execution time within the block's makespan, i.e.,*

$$\mathit{gas}_{\mathcal{T}}^{\mathit{TPM}}(tx) = \frac{t(tx)}{\sum_{tx'\in\mathcal{T}} t(tx')} \cdot v(\mathcal{T}).$$

In the two examples below, we assume $n \geq 2$ parallel threads and the optimal lock-based scheduler. Each transaction is written as $tx \simeq (t, O)$, where t is its execution time and O is the set of objects it accesses.

Example 5 (User Shill Attack on Shapley GCM)**.** *Let the set of other transactions be* $\{tx_1, tx_2, tx_3\}$ *with* $tx_1 \simeq (1, \{o_1\})$, $tx_2 \simeq (1, \{o_1\})$, $tx_3 \simeq (2, \{o_2\})$. *A user's* original *transaction is* $tx_4 \simeq (3, \{o_1\})$. $\mathcal{T} = \{tx_1, tx_2, tx_3, tx_4\}$

Baseline. *Under the Shapley GCM,* $\mathit{gas}_{\mathcal{T}}^{\mathit{Shapley}}(tx_4) = \frac{8}{3}$.

User Shill Attack. *If the user also submits a* fake $tx_5 \simeq (1, \{o_2\})$, *then:* $\mathit{gas}_{\mathcal{T}\cup\{tx_5\}}^{\mathit{Shapley}}(tx_4) = \frac{137}{60}, \mathit{gas}_{\mathcal{T}\cup\{tx_5\}}^{\mathit{Shapley}}(tx_5) = \frac{11}{30}$. *Hence, the user's* total *charge is* $\frac{137}{60} + \frac{11}{30} = \frac{53}{20} < \frac{8}{3}$, *Thus, by sending two transactions, the user reduced the gas used.*

Example 6 (Scheduler Shill Attack on TPM GCM)**.** *Let the set of other transactions be* $tx_1 \simeq (6, \{o_1\})$, $tx_2 \simeq (6, \{o_2\})$.

Baseline. *Under the TPM GCM we have that the gas of both transactions is 3, i.e.,* $\mathit{gas}_{\mathcal{T}}^{\mathit{TPM}}(tx_1) = 3$ *and* $\mathit{gas}_{\mathcal{T}}^{\mathit{TPM}}(tx_2) = 3$.

Scheduler Shill Attack. *If the scheduler submits a* fake $tx_3 \simeq (6, \{o_1, o_2\})$, *then* $\mathit{gas}_{\mathcal{T}}^{\mathit{TPM}}(tx_1) = \mathit{gas}_{\mathcal{T}}^{\mathit{TPM}}(tx_2) = \mathit{gas}_{\mathcal{T}}^{\mathit{TPM}}(tx_3) = 4$. *Hence, the total gas received by the scheduler becomes* $12 - 4 = 8 > 6$. *Thus, by sending a fake transaction, the scheduler earns more gas.*

6.2 Shill-proofness

Taking a step back, we look at why these attacks happened and what properties are required in addition to those mentioned in [2]. For a user shill attack, the user was able to insert a fake transaction (or transactions) to reduce the gas used strategically. Denoting it in mathematical terms, if the set of transactions is $\mathcal{T}$, and the user's transaction is $tx_1 \in \mathcal{T}$, then if there exists a $\mathcal{T}'$, such that

$$\mathsf{gas}_{\mathcal{T}}(tx_1) > \mathsf{gas}_{\mathcal{T}\cup\mathcal{T}'}(tx_1) + \mathsf{gas}_{\mathcal{T}\cup\mathcal{T}'}(\mathcal{T}'),$$

then, sending the fake transactions $\mathcal{T}'$ reduces gas usage.

Based on this, we introduce the following property that the GCM must satisfy to prevent user shill attacks.

Definition 14 (User Shill Proofness)**.** *No user should be able to reduce the gas paid by their transaction by introducing more transactions, or* $\forall\, tx_1, \mathcal{T}, \mathcal{T}'$ *then:*

$$gas_{\mathcal{T}}(tx_1) \leq gas_{\mathcal{T}\cup\mathcal{T}'}(tx_1) + gas_{\mathcal{T}\cup\mathcal{T}'}(\mathcal{T}'). \tag{4}$$

Consider the Set Inclusion property (Property 4, Acilan et al. [2]), which states that if $T_1 \subseteq T_2$ then: $\mathsf{gas}_{T\cup T_1}(T_1) \leq \mathsf{gas}_{T\cup T_2}(T_2)$. User shill proofness is a subset of this property, i.e., if a GCM satisfies Set Inclusion, then it necessarily satisfies User Shill Proofness. However, the implications are much different. While Set Inclusion tries to counter user collusion, User Shill Proofness deals with users' fake transactions to reduce cost. While Set Inclusion is a good-to-have property, User Shill Proofness is a fundamental requirement to avoid throughput reduction due to fake transactions. We therefore view shill-proofness as a *new* robustness notion rather than a restatement of coalition- or collusion-proofness for multi-dimensional fee markets [2, 22]. The gas-only Set Inclusion property asks how a single transaction's gas compares as its declared set grows; shill-proofness instead asks whether splitting a transaction into several (otherwise useless) transactions can lower the total charge. The gap widens once we move from gas to *fees* in the presence of contingency (Section 7): our fee-based notions additionally account for the priority fee, the burned portion of the fee, and shills that under-execute or fail, none of which the gas-only property captures.

Next, we analyze the reasons behind scheduler shill attacks. The scheduler can add a transaction in a natural GCM and increase the gas paid by the users, effectively increasing their revenue. Denoting it mathematically, if the set of transactions submitted is $\mathcal{T}$, then if there exists $\mathcal{T}'$ such that $\mathsf{gas}_{\mathcal{T}}(\mathcal{T}) < \mathsf{gas}_{\mathcal{T}\cup\mathcal{T}'}(\mathcal{T})$, then the scheduler can add fake transactions in $\mathcal{T}'$ to increase the total gas received by it.

To prevent this, we introduce scheduler shill proofness.

Definition 15 (Scheduler Shill Proofness)**.** *No scheduler should be able to increase the gas paid by honest transactions by introducing more transactions, or*

$$gas_{\mathcal{T}}(\mathcal{T}) \geq gas_{\mathcal{T}\cup\mathcal{T}'}(\mathcal{T}). \tag{5}$$

Interestingly, we find that Scheduler Shill Proofness is at odds with a seemingly unrelated property, Efficiency, which was suggested by Acilan et al. [2].

Definition 16 (Efficiency)**.** *The sum of gas used by all transactions in the schedule must equal the makespan of the schedule, i.e., if* $v(\mathcal{T})$ *represents the makespan of the schedule with transactions* $\mathcal{T}$*, then* $gas_{\mathcal{T}}(\mathcal{T}) = v(\mathcal{T})$.

Lemma 3 (Scheduler Shill Proofness vs Efficiency)**.** *It is impossible for a GCM with a limited number of parallel cores to achieve both Scheduler Shill Proofness and Efficiency.*

Proof. Consider n cores and $n+1$ completely parallel transactions, which take time t on a single core to execute. Any subset of n transactions can be scheduled in parallel such that the makespan is t, but the $(n+1)$-th transaction must be added sequentially, such that the makespan becomes $2t$. Scheduler Shill Proofness would require that the $(n+1)$-th transaction consume at least t gas. Since all transactions could be the $(n+1)$-th transaction, all transactions must consume t gas, and that violates the efficiency requirement that states the total gas consumed must be the makespan. □

Note that the proof relies on atomicity of the transaction, and if the transaction execution can be distributed across multiple cores, then this proof would not work. This is because the sequential transaction can be divided into smaller chunks and passed between cores to complete the execution in $\frac{n+1}{n} \cdot t$ time. This is further discussed in Section 9.

While it may be tempting to conclude that if a Gas Computation Mechanism (GCM) satisfies the two properties defined above then it is resistant to shill attacks, thereby yielding throughput optimality (if the schedule is optimal), this conclusion is not correct. In practice, most blockchain protocols also require users to specify a priority fee as discussed in Section B. Combined with the execution contingency as introduced in Section 4, shill attacks can potentially be made much worse by the adversary. The scheduling objective proposed in [2], which focuses on minimizing makespan, does not capture this priority structure. We therefore argue that the objective should instead be to maximize utility for scheduler rather than minimize makespan.

The dependence on both fee and gas consumption is critical. A transaction may consume substantial gas while offering only a negligible gas price, creating opportunities for profitable shill attacks. For instance, suppose a transaction offers a gas price of only $\varepsilon \sim 0$, yet its gas consumed is determined by one of the GCM methods described in [2]. A recurring feature of these methods is that adding a highly parallelizable transaction to the schedule reduces the effective gas consumption attributed to other transactions. A user can therefore submit a low-priced, highly parallelizable fake transaction to decrease the gas consumption of their actual transaction, thereby lowering the total fee paid.

To address this vulnerability, we follow by refining the notion of shill-proofness (and related properties), formulating them in terms of paid fees rather than raw gas consumption.

7 Fee-based Shill Proofness

We have so far analyzed shill attacks on GCMs through the lens of *gas consumed*. Now, we revisit shill attacks with a focus on the *fee paid* in the presence of *contingent transactions*. When a contingent transaction under-executes, it could pay a much lower fee compared to the fee it *could* have paid if it had executed fully. This asymmetry enables new

fee-reduction strategies: rational users can inject failing or near-failing transactions to alter contention and scheduling so that their high-value transaction pays strictly less. We formalize fee-based shill-proofness, show why baseline designs typically violate it when failures pay near-zero, and outline conditions that restore robustness.

We begin by first discussing the goal of the next two sections. When considering how to compute gas (GCM) or the fee paid by the transaction (TFM) in the context of parallel execution, the first question that rises is, what schedule must be used? Finding the best schedule $\mathcal{S}$ that optimizes scheduler revenue constitutes our first aim. However, optimizing the revenue requires us to also know what fee a transaction pays in the schedule, and thus while computing the schedule, the fee paid by the transaction based on the said schedule and the priority fee set by the user, must also be computed in conjunction. Moving forward, we denote the schedule by $\mathcal{S}$, since given the fee-structure, all parties can deterministically calculate it. Also, the sub-schedule is represented by $\mathcal{S}_{tx}^{\mathcal{T}}$. Adversaries are assumed to not want to change the execution of any transaction, i.e., $\text{Exec}(s \xrightarrow{\mathcal{S}_{tx}^{\mathcal{T}}} s', tx_1) = \text{Exec}(s \xrightarrow{\mathcal{S}_{tx}^{(\mathcal{T}\cup\mathcal{T}')}} s'', tx_1)$. For brevity, denote $\mathcal{S}(\mathcal{T})$ by $\mathcal{S}$, and $\mathcal{S}(\mathcal{T}\cup\mathcal{T}')$ as $\mathcal{S}'$.

User shill-proofness (Definition 14) ensures that users would not be able to reduce the gas or computation units attributed to their transactions by adding shill transactions. Given a model in which contingent transactions exist, an adversarial user must not be able to add *fake* contingent transactions that always under-executes to reduce the *fee* paid by its main transaction. Formally,

Definition 17 (Fee-Based User Shill Proofness)**.** *Introducing fake transactions cannot reduce the total fee paid by the expected execution of transaction tx_1, even if while under-executing the fake transactions have a zero user-ideal fee: $f_{\text{ui}} = 0$. That is, for any state s and $\forall\, tx_1, \mathcal{T}, \mathcal{T}'$*

$$f_{\text{act}}(s,\mathcal{S},tx_1) \le f_{\text{act}}(s,\mathcal{S}',tx_1) + f_{\text{act}}(s,\mathcal{S}',\mathcal{T}') \tag{6}$$

Here, we assume that tx_1 never under-executes and always pays the attainable fee f_{att}. For the three risk divisions defined in Section 5, this translates to different conditions.

- **User-Friendly Division**

$$f_{\text{att}}(s,\mathcal{S},tx_1) \le f_{\text{att}}(s,\mathcal{S}',tx_1)$$

- **Scheduler-Friendly Division**

$$f_{\text{att}}(s,\mathcal{S},tx_1) \le f_{\text{att}}(s,\mathcal{S}',tx_1) + f_{\text{att}}(s,\mathcal{S}',\mathcal{T}')$$

- **Even-Steven Division**

$$f_{\text{att}}(s,\mathcal{S},tx_1) \le f_{\text{att}}(s,\mathcal{S}',tx_1) + 0.5\cdot f_{\text{att}}(s,\mathcal{S}',\mathcal{T}')$$

- **Generalized Division**

$$f_{\text{att}}(s,\mathcal{S},tx_1) \le f_{\text{att}}(s,\mathcal{S}',tx_1) + \alpha\cdot f_{\text{att}}(s,\mathcal{S}',\mathcal{T}')$$

Next, we look at re-defining scheduler shill proofness. Here, we relax the assumption that the scheduler utility is defined in terms of fee paid, since the scheduler is introducing its own transactions, and it only receives back the fee received back from, thus incurring a cost of $f_{\text{act}} - r_{\text{act}}$ from the transaction. From Definition 15, a scheduler must not be able to strategically insert fake transactions such that the fee paid by other transactions increases more than the cost it pays.

Definition 18 (Fee-Based Scheduler Shill Proofness)**.** *No scheduler should be able to increase the revenue received from honest transaction (tx_1) by more than the cost incurred by introducing fake transactions ($\mathcal{T}'$), even if all transactions in $\mathcal{T}'$ under-execute. In other words, $\forall\, tx_1, \mathcal{T}, \mathcal{T}'$*

$$\begin{aligned} &r_{\text{act}}(s,\mathcal{S}',tx_1) - r_{\text{act}}(s,\mathcal{S},tx_1) \\ &\quad \le f_{\text{act}}(s,\mathcal{S}',\mathcal{T}') - r_{\text{act}}(s,\mathcal{S}',\mathcal{T}') \end{aligned} \tag{7}$$

We assume that $r_{\text{act}} = \gamma\cdot f_{\text{act}}$, and thus,

$$\gamma(f_{\text{act}}(s,\mathcal{S}',tx_1) - f_{\text{act}}(s,\mathcal{S},tx_1)) \le (1-\gamma) f_{\text{act}}(s,\mathcal{S}',\mathcal{T}') \tag{8}$$

Again, we assume that tx_1 never under-executes and always pays the attainable fee f_{att}, and all introduced shill transactions under-execute to pay as little as possible. For the risk divisions of Section 5, this translates to different conditions.

- **User-Friendly**

$$f_{\text{att}}(s,\mathcal{S}',tx_1) - f_{\text{att}}(s,\mathcal{S},tx_1) \le 0$$

- **Scheduler-Friendly**

$$f_{\text{att}}(s,\mathcal{S}',tx_1) - f_{\text{att}}(s,\mathcal{S},tx_1) \le \frac{1-\gamma}{\gamma} f_{\text{att}}(s,\mathcal{S}',\mathcal{T}')$$

- **Even-Steven Division**

$$f_{\text{att}}(s,\mathcal{S}',tx_1) - f_{\text{att}}(s,\mathcal{S},tx_1) \le \frac{1-\gamma}{2\gamma} f_{\text{att}}(s,\mathcal{S}',\mathcal{T}')$$

- **Generalized Division**

$$f_{\text{att}}(s,\mathcal{S}',tx_1) - f_{\text{att}}(s,\mathcal{S},tx_1) \le \alpha\frac{1-\gamma}{\gamma} f_{\text{att}}(s,\mathcal{S}',\mathcal{T}')$$

Based on the above feasible regions, we prove Theorem 3, which implies that transaction fees must not depend on parallelism if a user friendly mechanism is used.

Theorem 3 (Independence in User-Friendly Mechanisms)**.** *In a parallel execution environment using a User-Friendly risk division (where User Risk $\text{UR} = 0$ and $\alpha = 0$), a Transaction Fee Mechanism (TFM) satisfies both Fee-Based User Shill Proofness and Fee-Based Scheduler Shill Proofness if and only if the attainable fee of a transaction is independent of all other transactions in the schedule.*

The proof follows from the User-Friendly risk division's conditions for Fee-Based User and Scheduler Shill Proofness.

Proof. Under user-friendliness, User Shill Proofness implies:

$$f_{\text{att}}(s, \mathcal{S}, tx_1) \leq f_{\text{att}}(s, \mathcal{S}', tx_1)$$

And, scheduler shill proofness gives:

$$f_{\text{att}}(s, \mathcal{S}', tx_1) \leq f_{\text{att}}(s, \mathcal{S}, tx_1)$$

The two can be satisfied together only when:

$$f_{\text{att}}(s, \mathcal{S}', tx_1) = f_{\text{att}}(s, \mathcal{S}, tx_1)$$

And, this implies that the fee paid by the user is independent of other transactions in the system.

□

Let's revisit the running example to illustrate shill attacks.

Example 7 (Running Example: Shilling Alice's Swap)**.** *The two attacks apply to Alice's swap. In a* user *shill attack, Alice adds a contingent transaction touching many idle objects, disjoint from* $\{o_{\mathsf{px}}, o_{\mathsf{pool}}, o_{\mathsf{bal}}\}$*, but does not pay the fee since the transaction never uses the declared objects. Because the throwaway runs in parallel with* tx_{A}*, the GCM or the TFM associated with it, now spreads the block's makespan over more transactions and attributes* less *gas to* tx_{A}*, lowering Alice's fee. In a* scheduler *shill attack, the scheduler inserts a transaction that also touches the in-demand pool* o_{pool}*; this makes* tx_{A} *look like it contends for more shared state, inflating the gas charged to it. Both manipulations change Alice's fee without changing what her transaction actually does.*

8 Transaction Fee Mechanism Design Space

We now formalize the design space of TFMs for parallel execution environments and identify the key properties such mechanisms must satisfy. Our goal is to understand how different choices of *schedulers* and *risk-divisions* interact, and to characterize which combinations yield mechanisms that are both economically robust and resistant to manipulation.

At a high level, a TFM operates in conjunction with a scheduler that determines a feasible execution schedule for a set of transactions, subject to parallelism and object constraints. The fee mechanism must determine how execution uncertainty is priced, leading to different *risk-division* regimes as defined in Section 5.1, ranging from *user-friendly* (users pay only for objects they actually consume), to *scheduler-friendly* (users pay for all objects they declare), with intermediate compromises generalized by the generalized division rule.

Further, when creating the schedule, the scheduler does not know the execution trace of the transaction, and thus cannot know the objects that would be used by the transaction. As defined in Section 5.2, the scheduler could have various utility definitions as well when designing the schedule, which ranges from *optimistic*, *pessimistic* as well as somewhere in between. We would look at how to create schedules based on only *optimistic* and *pessimistic* schedulers and leave a generalized utility function as a future work.

A central difficulty in designing TFMs in this setting is the inherently *cyclic* dependence between scheduling and fees. On the one hand, the scheduler's choice of which transactions to include and how to order or parallelize them depends on the fees offered by transactions. On the other hand, the fee paid by a transaction may itself depend on the realized execution schedule, particularly when fees are tied to the realized parallelism. This circular dependence complicates both the analysis and the implementation of TFMs.

8.1 Object-Weighted TFM (OW-TFM)

To break this cyclic dependency, we begin by focusing on a class of transaction fee mechanisms inspired by the weighted-area gas model introduced in prior work on transaction fee market design for parallel execution [2]. In this model, the fee charged to a transaction is *independent of the execution schedule*, while still capturing the effects of parallelism by pricing the objects based on past executed transactions. A key assumption for OW-TFM is that object demand and thus object prices, remain relatively stable across consecutive rounds, allowing transactions to be priced using per-object prices that are fixed at the time of inclusion.

An important consequence of using past transaction data to determine the fee paid by the transaction is that *no shill attacks* are relevant unless the price is manipulated by inserting transactions in previous blocks.

Concretely, we assume that a transaction tx uses computation units t and declares a set of objects it may use during execution. Each object o has a price p_o. If tx uses a object o and declares a priority fee of $\pi \geq 1$, then it pays a price $\pi \cdot p_o \cdot t$ for that object. To capture execution uncertainty, we parameterize the division of risk between users and the scheduler by a parameter $\alpha \in [0,1]$. For any object that is *declared but not used* during execution, the transaction pays a fee of $\alpha \cdot p_o$. Setting $\alpha = 0$ corresponds to user-friendly risk division, while $\alpha = 1$ corresponds to scheduler-friendly risk division. Intermediate values of α interpolate between these extremes and allow for more balanced risk-sharing arrangements.

Example 8 (Running Example: Pricing Alice's Swap)**.** *Under OW-TFM, with Alice's priority* $\pi = 1$*, compute* $t = 1$*, and object prices* $p_{o_{\mathsf{px}}} = 1$*,* $p_{o_{\mathsf{pool}}} = 3$*,* $p_{o_{\mathsf{bal}}} = 1$*, the favorable execution touches all three objects and costs* $1+3+1 = 5$*. On the under-executed branch she uses only* o_{px}*, but still owes* α *times the price of the two declared-but-unused objects, so she pays* $1+\alpha(3+1) = 1+4\alpha$*. This recovers the same numbers as before:* 1 *at* $\alpha = 0$*,* 3 *at* $\alpha = 1$*, and* 5 *at* $\alpha = 1$*, confirming that OW-TFM realizes the risk split of Section 5 with a fee that does not depend on the rest of the schedule.*

We use a price update function similar to EIP-1559 [28]. In order to formally describe this function, define the realized utilization of object o in block b (with transactions B_b) as

$$\mathcal{U}_{o,b} := \sum_{tx:=(t,_,R,W)\in B_b} t \cdot \mathbf{1}\{o \in \{R \cup W\}\},$$

which represents the time for which the object was reserved in the block . For each object $o \in O$, we fix a target utilization $\mathcal{U}_o^\star > 0$. Let $p_{o,b} > 0$ denote the posted price for objects o in block b. Fix a responsiveness parameter $\eta > 0$. The price for the next block is updated according to:

$$\begin{aligned} p_{o,(b+1)} &= p_{o,b} \cdot \exp\left(\eta \cdot \frac{\mathcal{U}_{o,b} - \mathcal{U}_o^\star}{\mathcal{U}_o^\star}\right) \\ &\approx p_{o,b} \cdot \left(1 + \eta \cdot \frac{\mathcal{U}_{o,b} - \mathcal{U}_o^\star}{\mathcal{U}_o^\star}\right). \end{aligned}$$

Shill-proofness of the OW-TFM. Here, prices are set from the measured object utilization $\mathcal{U}_{o,b}$ of the *previous* block. Adding a transaction can only add to this measured usage or leave it unchanged: each declared object contributes a non-negative amount, and a contingent transaction that under-executes simply contributes less (possibly zero), but never a negative amount. So a shill can push the recorded usage of an object *up* but cannot push it *down*. This is why a contingent transaction, even though it may under-execute, cannot be used to *lower* an object's measured consumption: under-execution only means the shill itself counts for less, not that honest usage is erased. The one way to reduce an object's recorded usage would be to remove honest transactions that use it (censorship), which is a separate kind of manipulation that we treat as out of scope. Consequently, the only relevant manipulation here is a scheduler shill attack that inflates an object's usage to raise its price in the next block.

Lemma 4 (Scheduler Shill Proofness of OW-TFM)**.** *Consider block B_b with a block number b and a shill transaction tx^s which declares object o (i.e., $o \in R \cup W$). Let $\mathcal{U}_o^\star$ represent the object's target and average usage. If $\alpha \geq \pi_o \eta \frac{\gamma}{(1-\gamma)}$, then the resulting TFM is scheduler shill proof, where π_o is the average priority for transactions using o in block $b+1$.*

Proof. Since this is a multi-block attack, assume that the state is common before block b, and that tx_1 is a transaction in block $b+1$ (or all transactions that declare o, combined into a single transaction), that has t_{tx_1} of $\mathcal{U}_o$. The priority for the transaction is the average priority of all transactions in this block that use object o and is represented by π_o.

Without Shill Transaction. In block b, without the shill transaction, let the object utilization is $\mathcal{U}_{o,b}$. The price of tx_1 with priority π_i without any manipulation in block $b+1$ is:

$$f_{\text{att}}(s, \mathcal{S}, tx_1) = p_{o,b} \cdot \left(1 + \eta \cdot \frac{\mathcal{U}_{o,b} - \mathcal{U}_o^\star}{\mathcal{U}_o^\star}\right) \cdot \pi_o \cdot \mathcal{U}_o^\star$$

With Shill Transaction. In block b, let the scheduler add the shill transaction tx^s. Since the priority would only increase the fee the shill transaction pays, we will assume that the priority fee is 1. The cost of tx_1 is now given by

$$\begin{aligned} f_{\text{att}}(s, \mathcal{S}', tx_1) &= p_{o,b} \cdot \left(1 + \eta \cdot \frac{\mathcal{U}_{o,b} + t_s - \mathcal{U}_o^\star}{\mathcal{U}_o^\star}\right) \cdot \pi_o \cdot \mathcal{U}_o^\star \\ &= f_{\text{att}}(s, \mathcal{S}, tx_1) + \eta p_{o,b} \pi_o t_s, \end{aligned}$$

whereas the (attainable) fee paid by the shill transaction is

$$f_{\text{att}}(s, \mathcal{S}', tx_s) = \pi_s \cdot p_{o,b} t_s \geq p_{o,b} t_s$$

From fee-based scheduler shill proofness,

$$\begin{aligned} f_{\text{att}}(s, \mathcal{S}', tx_1) - f_{\text{att}}(s, \mathcal{S}, tx_1) &\leq \alpha \frac{1-\gamma}{\gamma} f_{\text{att}}(s, \mathcal{S}', tx_s) \\ \eta p_{o,b} \pi_o t_s \leq \alpha \frac{1-\gamma}{\gamma} p_{o,b} t_s &\implies \eta \pi_o \frac{\gamma}{(1-\gamma)} \leq \alpha \end{aligned}$$

□

8.2 Empirical Parameterization

We now plug in real-world numbers from Ethereum to see what our theoretical bounds look like in practice.

Priority-to-base Fee Ratio (π_o). On Ethereum, the average priority fee is about 4× the base fee, giving $\pi_o = 5$ (since π_o includes the base fee itself). The distribution is skewed: most transactions have a low priority multiplier (median ≈ 1.2), but a small fraction pay much more (the 95th percentile exceeds 16). These values are obtained from the Dune Analytics query in Open Science. We evaluate three scenarios: $\pi_o \in \{1.2, 5, 16\}$, representing median, mean, and high-priority transactions.

Responsiveness (η) and Retention Ratio (γ). EIP-1559 [28] sets $\eta = \frac{1}{8}$ as the price update speed. The retention ratio $\gamma = \frac{r}{f}$ is the scheduler's fee share, with the rest burned. Let $\gamma = \frac{1}{2}$ throughout, i.e., half of each fee is burned.

Scheduler Shill-proofness Threshold. By Lemma 4, OW-TFM is scheduler shill proof when $\alpha \geq \pi_o \cdot \eta \cdot \frac{\gamma}{1-\gamma}$. Plugging in $\eta = \frac{1}{8}$ and $\gamma = \frac{1}{2}$ gives $\alpha_{\min} = \frac{\pi_o}{8}$. Table 2 summarizes the result for each scenario. At the median priority, a small amount of user risk ($\alpha = 0.15$) is enough to make shill attacks unprofitable. At the mean priority, the required α rises to 0.625, which falls between Even-Steven and scheduler-friendly. At the 95th percentile, $\alpha_{\min}$ exceeds 1, meaning no risk division within our framework can guarantee shill-proofness against the highest-priority attackers. However, since π_o represents the *average* priority across all transactions using a given object, the mean value of 5 is the most representative case for a typical block.

Contingency Impact. Heimbach et al. [21] find that 42.6% of Ethereum transactions would have benefited from access lists,

Table 2: Minimum risk-division parameter α for scheduler shill-proofness of OW-TFM under empirically-estimated Ethereum fee parameters.

Scenario	π_o	γ	$\alpha_{\min}$
Median priority	1.2	0.5	0.15
Mean priority	5	0.5	0.625
95th percentile	16	0.5	—

and that 19.6% of the access lists that were supplied turned out to be wrong, mostly because the state at execution time was not known in advance (which we prove in Theorem 1). These numbers show that contingency is common in practice. Under a scheduler-friendly mechanism ($\alpha = 1$), users with inaccurate declarations overpay for objects they never use. Under a user-friendly mechanism ($\alpha = 0$), the scheduler loses revenue instead. Since roughly 40% of transactions are involved in conflicts [20, 32] and a large share of those exhibit contingent behavior, the choice of α has a real economic impact that grows with block size and object contention.

9 Conclusion and Discussions

Parallel execution promises throughput gains for modern blockchains, but it fundamentally complicates the economics of transaction inclusion. In this paper, we show that when execution is uncertain, parallel execution based transaction fee mechanisms face inherent limitations. Execution contingency introduces a structural trade-off between user risk and scheduler risk: without pre-execution or collapsing to constant fees, no mechanism can simultaneously eliminate both. This impossibility is not an artifact of particular designs, but follows from the inherent sequentiality of determining object usage in expressive smart contracts. As a result, fee mechanisms must make explicit choices about how uncertainty is allocated, rather than attempting to hide it.

Independently, we demonstrate that parallel gas computation mechanisms are vulnerable to rational shill attacks by both users and schedulers, even when they satisfy previously proposed fairness and efficiency properties. We formalize fee-based shill-proofness as a necessary security requirement and show how execution contingency amplifies these attacks by reducing the cost of fake transactions. Taken together, our results clarify that pricing parallel execution is a mechanism-design problem with unavoidable trade-offs. Within this constrained design space, we show that object-weighted fee mechanisms with appropriate risk-division parameters can achieve robustness against shill attacks while remaining compatible with parallel execution. These findings provide foundational guidance for evaluating and designing transaction fee mechanisms in emerging parallel blockchain systems.

On Atomicity of Scheduled Transaction Using today's systems, the number of parallel processes can exceed the number of total cores that the system possesses. This then leads to a distribution of workload across all cores through (very optimized) context switching of processes across all cores. In such a case, the actual number of threads available for execution can be considered to be infinite, but the total time to execute each transaction is scaled by the extra number of threads in use at each time. In such a case, consider the example in the proof for Lemma 3. The $n+1$ transactions can now be scheduled on n threads, and due to context switching all the threads would share the workload. This leads to a makespan of $\sim \frac{n+1}{n}t$, which then can still satisfy efficiency, while satisfying scheduler shill-proofness. This can be useful especially with unsafe mode, where the locks on objects are not checked at any moment. (Unsafe mode is safe since the schedule created never has shared overlapping objects)

On Attainable Fee. We define the *attainable fee* as the fee a transaction would pay if all declared contingent objects were used during execution. This abstraction simplifies analysis by providing a clean upper bound on what a scheduler might reasonably expect to collect from a transaction.

In practice, however, some transactions may never use all declared contingent objects under any execution path. For example, consider an AMM router transaction that inspects multiple liquidity pools and conditionally routes execution to the one pool offering the best price. While such a transaction may declare several pools as contingent objects, only one will be accessed. In this case, the attainable fee does not correspond to any realizable execution. We intentionally abstract away this nuance. For the scheduler, distinguishing between "logically impossible" and "unlikely" execution paths is infeasible without executing the transaction.

On Execution of Transactions. Our framework treats execution outcomes adversarially or pessimistically. Incorporating probabilistic models of execution, informed by historical behavior or application-level semantics, could enable mechanisms that optimize expected risk rather than worst-case risk. Such approaches would necessarily introduce new assumptions and attack surfaces, and merit careful security analysis.

Declaring Contingent Objects. Our mechanisms require users to declare the objects that their transaction may touch, including contingent ones. In practice, declarations can be produced by client software, the same way Ethereum access lists are generated by simulating transactions before broadcasting them. Lying in either direction can be discouraged. If a user *under*-declares, i.e., touches an object it did not declare, this can be caught at execution time, since the runtime sees the access. The transaction can then be reverted while still being charged, as already happens for failed Ethereum transactions. If a user *over*-declares, i.e., lists objects it never uses, this can be made costly by charging for declared-but-unused objects, which is exactly what the scheduler-friendly division (and any $\alpha > 0$) does. Getting users to reveal their true object

sets is an important deployment constraint, and a mechanism that fully enforces it is left to future work.

Relation to Ethereum and EIP-1559. Ethereum today uses EIP-1559 [28], which prices a single, one-dimensional notion of gas and does *not* charge transactions for contention over specific objects. As long as execution is serial, there is no parallelism to price, so contention and contingency do not show up in the fee. The point of this paper is that *once* parallel execution is adopted, these effects appear and must be priced, otherwise the shill attacks and risk trade-offs we describe become possible. This is why the setting is relevant to chains that already run transactions in parallel, such as Sui [11], Solana [10], and Monad [26], and to Ethereum's own roadmap toward parallel execution via block-level access lists [33].

Acknowledgements

We would like to thank Alberto Sonnino (Mysten Labs), Kushal Babel (Category Labs) and Alexander Spiegelman (Aptos) for discussions on the project, and helping us understand the landscape of the various deployed blockchains.

Ethical Considerations

Stakeholders and Scope. This work primarily affects blockchain protocol designers, system architects, and researchers developing execution engines and transaction fee mechanisms for parallel execution. Indirect stakeholders include blockchain users whose transaction fees and execution outcomes depend on these mechanisms, validators or schedulers whose incentives are shaped by fee design, and application developers whose smart contracts interact with parallel execution environments. As transaction fee mechanisms influence access to blockspace, the results are relevant to auditors, regulators, and ecosystem participants concerned with fairness and manipulation resistance in blockchain infrastructure.

Research Conduct. The research is theoretical and analytical, supported by formal modeling and adversarial reasoning. We analyze fee mechanisms under explicitly stated rational-adversary models and do not involve experiments on live systems, user data, or deployed protocols. Prior work is cited carefully, and our contributions are framed as identifying fundamental limitations and design trade-offs rather than prescribing immediate deployment recommendations.

Benefits and Public Interest. By identifying impossibility results and previously unrecognized attack vectors in parallel transaction fee mechanisms, this work helps prevent unsafe or economically unsound protocols from being deployed. Clarifying the trade-offs between user risk, scheduler risk, and shill resistance contributes to more transparent and accountable fee-market design, supporting fairer transaction inclusion and more predictable user costs. These insights serve the public interest by strengthening the robustness of blockchain systems that increasingly underpin financial and digital infrastructure.

Risks and Limitations. The techniques analyzed in this paper, such as shill attacks exploiting parallelism and execution uncertainty, could potentially be misused by adversarial actors to extract value from poorly designed systems. Profitable shills can waste scarce execution capacity, distort prices, and change inclusion incentives. Because the paper analyzes mechanism-level issues in [2]'s proposed designs and not vendor-specific implementation vulnerabilities, vendor disclosure is not directly applicable. Our analysis assumes rational, economically motivated adversaries and does not model fully malicious behavior, denial-of-service attacks, or off-chain collusion. We explicitly document these assumptions, and caution that deploying fee mechanisms outside their modeled conditions may lead to unintended consequences.

Open Science

The query below can be run as is on Dune analytics [9] to get the current priority to base ratio.

Listing 1: Query used to estimate priority-to-base fee ratios over the last 30 days.

```
WITH tx_fees AS (
  SELECT
    t.block_time,
    t.priority_fee_per_gas,
    b.base_fee_per_gas,
    CAST(t.priority_fee_per_gas AS DOUBLE) /
      NULLIF(CAST(b.base_fee_per_gas AS DOUBLE), 0)
      AS priority_to_base_ratio
  FROM base.transactions t
  INNER JOIN base.blocks b
    ON t.block_number = b.number
    AND t.block_date = b.date
  WHERE t.block_date >= CURRENT_DATE - INTERVAL
      '30' day
    AND b.base_fee_per_gas > 0
    AND t.priority_fee_per_gas IS NOT NULL
)
SELECT
  AVG(priority_to_base_ratio) AS
      avg_priority_to_base_ratio,
  APPROX_PERCENTILE(priority_to_base_ratio, 0.5)
    AS median_priority_to_base_ratio,
  MIN(priority_to_base_ratio) AS min_ratio,
  MAX(priority_to_base_ratio) AS max_ratio
FROM tx_fees;
```

References

[1] Mikerah A. and Sarisht Wadhwa. Block building is not just knapsack!, June 2024. Ethereum Research forum

post. URL: https://ethresear.ch/t/block-building-is-not-just-knapsack/19871.

[2] Bahar Acilan, Andrei Constantinescu, Lioba Heimbach, and Roger Wattenhofer. Transaction fee market design for parallel execution, 2025. URL: https://arxiv.org/abs/2502.11964, arXiv:2502.11964.

[3] aptos foundation. aip-27 · aptos-foundation/aips, 2022. URL: https://github.com/aptos-foundation/AIPs/blob/main/aips/aip-27.md.

[4] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. Mysticeti: Reaching the latency limits with uncertified dags. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL: https://www.ndss-symposium.org/ndss-paper/mysticeti-reaching-the-latency-limits-with-uncertified-dags/.

[5] Maryam Bahrani, Pranav Garimidi, and Tim Roughgarden. Transaction fee mechanism design with active block producers, 2023. URL: https://arxiv.org/abs/2307.01686, arXiv:2307.01686.

[6] Henry Berg and Todd A. Proebsting. Hanson's automated market maker. *Journal of Prediction Markets*, 3(1):45–59, April 2009. URL: https://ideas.repec.org/a/buc/jpredm/v3y2009i1p45-59.html.

[7] Hao Chung and Elaine Shi. Foundations of transaction fee mechanism design, 2022. URL: https://arxiv.org/abs/2111.03151, arXiv:2111.03151.

[8] Theo Diamandis, Alex Evans, Tarun Chitra, and Guillermo Angeris. Designing Multidimensional Blockchain Fee Markets. In *5th Conference on Advances in Financial Technologies (AFT 2023)*, volume 282 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.AFT.2023.4.

[9] Dune analytics. URL: https://dune.com/home.

[10] Solana Foundation. Sealevel - parallel processing thousands of smart contracts, Sep 2019. URL: https://solana.com/news/sealevel---parallel-processing-thousands-of-smart-contracts.

[11] SUI Foundation. Gas in sui | sui documentation, 2025. URL: https://docs.sui.io/concepts/tokenomics/gas-in-sui.

[12] Yotam Gafni. Deterring a small collusion is all you need. In *Proceedings of the ACM Web Conference 2026*, WWW '26, pages 31–39, New York, NY, USA, April 2026. Association for Computing Machinery. doi:10.1145/3774904.3792121.

[13] Yotam Gafni and Aviv Yaish. Greedy transaction fee mechanisms for (non-)myopic miners, 2022. URL: https://arxiv.org/abs/2210.07793v4.

[14] Yotam Gafni and Aviv Yaish. Barriers to collusion-resistant transaction fee mechanisms. In *Proceedings of the 25th ACM Conference on Economics and Computation*, EC '24, page 1074–1096, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3670865.3673469.

[15] Yotam Gafni and Aviv Yaish. Discrete and bayesian transaction fee mechanisms. In Stefanos Leonardos, Elise Alfieri, William J. Knottenbelt, and Panos Pardalos, editors, *Mathematical Research for Blockchain Economy*, pages 145–171, Cham, 2024. Springer Nature Switzerland.

[16] Yotam Gafni and Aviv Yaish. Scheduling with time discounts, 2024. doi:10.48550/arXiv.2402.08549.

[17] Yotam Gafni and Aviv Yaish. Transaction fee mechanisms robust to welfare-increasing collusion. *Games and Economic Behavior*, 157:351–375, March 2026. doi:10.1016/j.geb.2026.02.005.

[18] Pranav Garimidi, Lioba Heimbach, and Tim Roughgarden. Transaction fee mechanism design for leaderless blockchain protocols, 2025. URL: https://arxiv.org/abs/2505.17885, arXiv:2505.17885.

[19] Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press on Demand, Oxford, UK, 1995. doi:10.1093/oso/9780195085914.001.0001.

[20] Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, and Roger Wattenhofer. Defi and nfts hinder blockchain scalability. In Foteini Baldimtsi and Christian Cachin, editors, *Financial Cryptography and Data Security*, pages 291–309, Cham, 2024. Springer Nature Switzerland.

[21] Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, Roger Wattenhofer, and Patrick Züst. Dissecting the eip-2930 optional access lists. In Jeremy Clark and Elaine Shi, editors, *Financial Cryptography and Data Security*, pages 292–302, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-78676-1_16.

[22] Aggelos Kiayias, Elias Koutsoupias, Giorgos Panagiotakos, and Kyriaki Zioga. One-dimensional vs. Multidimensional Pricing in Blockchain Protocols, June 2025. arXiv:2506.13271 [cs]. doi:10.48550/arXiv.2506.13271.

[23] Nir Lavee, Noam Nisan, Mallesh Pai, and Max Resnick. Does Your Blockchain Need Multidimensional Transaction Fees?, April 2025. URL: http://arxiv.org/abs/2504.15438, doi:10.48550/arXiv.2504.15438.

[24] Vijay Mohan and Peyman Khezr. Blockchains, mev and the knapsack problem: A primer, 2024. URL: https://arxiv.org/abs/2403.19077, arXiv:2403.19077.

[25] Monad. Gas on monad | monad developer documentation. URL: https://docs.monad.xyz/developer-essentials/gas-on-monad#gas-price-detail.

[26] Monad. Parallel execution | monad developer documentation. URL: https://docs.monad.xyz/monad-arch/execution/parallel-execution.

[27] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. RainBlock: Faster transaction processing in public blockchains. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 333–347. USENIX Association, July 2021. URL: https://www.usenix.org/conference/atc21/presentation/ponnapalli.

[28] Ethereum Improvement Proposals. Eip-1559: Fee market change for eth 1.0 chain, April 2019. URL: https://eips.ethereum.org/EIPS/eip-1559.

[29] Tim Roughgarden. Transaction fee mechanism design. *J. ACM*, 71(4), August 2024. doi:10/g9z6nw.

[30] Alexander Spiegelman, Neil Giridhaman, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[31] Sui. Object Model | Sui Documentation, 2025. URL: https://docs.sui.io/concepts/object-model.

[32] Anton Wahrstätter. Execution dependencies, April 2025. URL: https://ethresear.ch/t/execution-dependencies/22150.

[33] Toni Wahrstätter, Dankrad Feist, Francesco D'Amato, Jochem Brouwer, Ignacio Hagopian, Felipe Selmo, Rahul, and Stefan, 2025. URL: https://eips.ethereum.org/EIPS/eip-7928.

[34] Aviv Yaish, Kaihua Qin, Liyi Zhou, Aviv Zohar, and Arthur Gervais. Speculative denial-of-service attacks in ethereum. In *33rd USENIX Security Symposium (USENIX Security 24)*, USENIXSEC '24, pages 3531–3548, Philadelphia, PA, 8 2024. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/yaish.

A Convergence Attacks

While a recent elegant work has shown that multidimensional fee mechanisms may be slower to converge to a steady state [22], the proof relies on a very strict assumption. Intuitively, the assumption boils down to considering very different convergence time distributions for single- and multidimensional mechanisms. In more detail, it is assumed that the difference between the distribution of convergence times of a single dimensional mechanism and of any given single resource of a multidimensional mechanism is upper-bounded by a single parameter. Thus, given enough resources, one is likely to sample a "bad" relative convergence time.

Now, we present a concrete counter-example highlighting the issues with the result of [22].

Proposition 1. *A sequence of transactions representing stable demand for each individual resource induces divergent and oscillatory behavior for a single-dimensional mechanism, while a multidimensional mechanism remains stable.*

Proof. We begin by fixing some multidimensional fee mechanism. The most reasonable single-dimensional "analog" for the multidimensional mechanism would set its gas limit to equal the lowest limit of any single resource of the multidimensional mechanism. To see why, note that a lower limit in the single-dimensional case does not make full use of the block-time, and higher limits result in easy-to-execute attacks.

Next, consider a sequence of transactions such that for each block, the mempool receives a number of transactions which equals the number of resources. Each of these transactions consumes exactly a single resource, where the amount that is consumed equals exactly the "target" amount of the resource. In the multidimensional world, this means that the prices would remain stable. The multidimensional case is thus trivially faster to converge. □

Moreover, it is unclear if convergence time is an appropriate benchmark. In fact, we devise a sequence of transactions where the "bad" outcome is for a mechanism to remain stable, as that indicates ignorance of the market's demand dynamics. Then, while a multidimensional mechanism would constantly be in flux, this is the *desired* behavior, stemming from a reaction to fine-grained changes in demand which the single-dimensional mechanism is ignorant of.

Example 9. *Consider two resources with a target of $\frac{G}{2}$ and a limit of G, and 3 transactions. For $i \in \{1,2\}$, let tx_i consume*

$\frac{G}{4} - \frac{\varepsilon}{2}$ *of* r_i*, and let* tx_3 *consume* $\frac{\varepsilon}{2}$ *of each of* r_1, r_2 *(i.e., a total of* $\frac{\varepsilon}{2}$ *is consumed). Moreover,* tx_1, tx_2 *find the two resources to be perfect substitutes, while* tx_3 *does not (e.g., consider two AMMs for the same token pair, an arbitrageur needs both, but simple traders with a large enough price-impact tolerance may not really have a preference). A single-dimensional mechanism would see a constant demand that equals the target (*$\frac{G}{2}$*), and so would remain stable. However, the multi-dimensional mechanism would understand that resources are under-utilized, and thus would lower fees.*

At a high level, understanding how markets react to multi-dimensional fee mechanisms and the corresponding impact on convergence times highlight an interesting question for future work to answer empirically. Ethereum's recent adoption of EIP-4844 could prove useful to such an endeavor; In particular, it seems like the variance in transaction fees has become lower since March 13, 2024, when the Dencun upgrade activated EIP-4844 on Ethereum's mainnet.

B Priority Schedules

We now analyze currently deployed priority-based TFM schedules under parallel execution. We show that these mechanisms exploit parallelism sub-optimally. That is because in the worst case, their *weighted throughput* (total fee collected) is the same as a linear schedule. We focus on two classes, GREEDY (fee-ordered partial orders over conflicts) and RANDOM (randomized ordering), and prove that in the worst case the weighted throughput of each is bounded by the number of threads available for parallel execution.

Transaction fees play a central role in determining the order and timing of transaction execution. The notion of schedulers that prioritize transactions based on some measure of "importance" is not new [13, 16]. Indeed, every major blockchain platform employs some variant of a *priority scheduler*, where transactions offering higher fees (or utility) are executed before those with lower ones. This mechanism reflects the rational principle that users willing to pay more should receive faster and more reliable inclusion.

Block-building based entirely on fees has often been compared to the well-studied *knapsack problem*, where the goal is to maximize total fee revenue under a limited computational or gas budget [1, 24]. Historically, however, the primary bottleneck in blockchain systems was *consensus latency* rather than *execution throughput*. Consequently, a greedy block-building heuristic (selecting transactions in decreasing order of fee per gas and executing them sequentially) became a natural choice, as in Bitcoin and Ethereum. We refer to such linear, fee-ordered scheduling as the *linear design*.

However, for modern blockchains like Solana, SUI, Aptos, Monad, the execution has become the bottleneck. Partial orders allow non-conflicting transactions to run concurrently while respecting fee-ordered precedence among conflicts. A representative instance is Sui's fee-ordered partial order over shared objects [11], built with a fast consensus (Mysticeti [4]). Transactions declare object usage (via Move Language [31]), enabling a fee-ordered DAG that preserves output equivalence while attempting parallel execution. This GREEDY policy seems rational as higher fee implies higher priority [13], but, can be revenue-suboptimal under congestion because early high-fee conflicts can block parallel opportunities elsewhere.

Example 10. *Consider* $\{tx_1, tx_2, tx_3, tx_4\}$ *with* $tx_1 \simeq (200, 4, \{o_1, o_3\})$, $tx_2 \simeq (150, 3, \{o_1, o_2\})$, $tx_3 \simeq (100, 2, \{o_2\})$, $tx_4 \simeq (100, 1, \{o_3\})$, *all writes, and block computation limit* $G = 400$. *Under GREEDY (fee-ordered DAG), precedence edges* $tx_1 \to tx_2$, $tx_2 \to tx_3$, *and* $tx_1 \to tx_4$ *yield* $tx_1 \to (tx_2, tx_4) \to tx_3$, *which exceeds G, dropping* tx_3. *Yet, an alternative valid schedule* $(tx_1, tx_3) \to (tx_2, tx_4)$ *respects G and admits all four, strictly improving revenue. Figure 1 illustrates the three schedules (linear, GREEDY, parallel optimum) and their object usage.*

The example demonstrates that fee-ordered schedules can waste available parallel compute which could have executed lower priority non-conflicting transactions.

For each schedule P, define $\alpha(P)$ as above. With an $\varepsilon \sim 0$ transaction priced at $\sim 1 + \varepsilon$ and a transaction with compute G priced at 1, the following lower bound demonstrates that GREEDY can be arbitrarily poor:

$$\alpha(\text{GREEDY}) = \frac{\varepsilon \cdot (1+\varepsilon)}{G} \approx 0.$$

Fortunately, in practice, systems cap transaction size at $T_{\max}$, leading to the following worst-case guarantee.

Lemma 5. *Given n parallel cores and* n_O *objects,*

$$\alpha(\textit{GREEDY}) = \frac{G - T_{max}}{G \cdot \min(n, n_O)},$$

and the lower bound is achievable in the limit.

Proof. Consider a single core machine, and enough transactions to fill the schedule in an optimal schedule. Consider the last scheduled transaction in GREEDY as tx. To get relative revenue, consider that this transaction pays 1 unit per computation used. Thus, any transaction scheduled must pay a gas price ≥ 1 and any transaction that fails to get scheduled pay a gas price ≤ 1. If the computation used by the schedule is $\leq G - T_{\max}$, then another transaction can be scheduled after GREEDY, which violates the definition of GREEDY. Thus, the computation used by the schedule is $> G - T_{\max}$, and the gas price is ≥ 1. This implies that the revenue received is $> G - T_{\max}$. Adding multiple cores cannot lower the revenue received, so the revenue generated is $R_{\text{GREEDY}} > G - T_{\max}$.

The best possible revenue from OPT would be when the complete block is scheduled with as many transactions in parallel as possible. This would include the

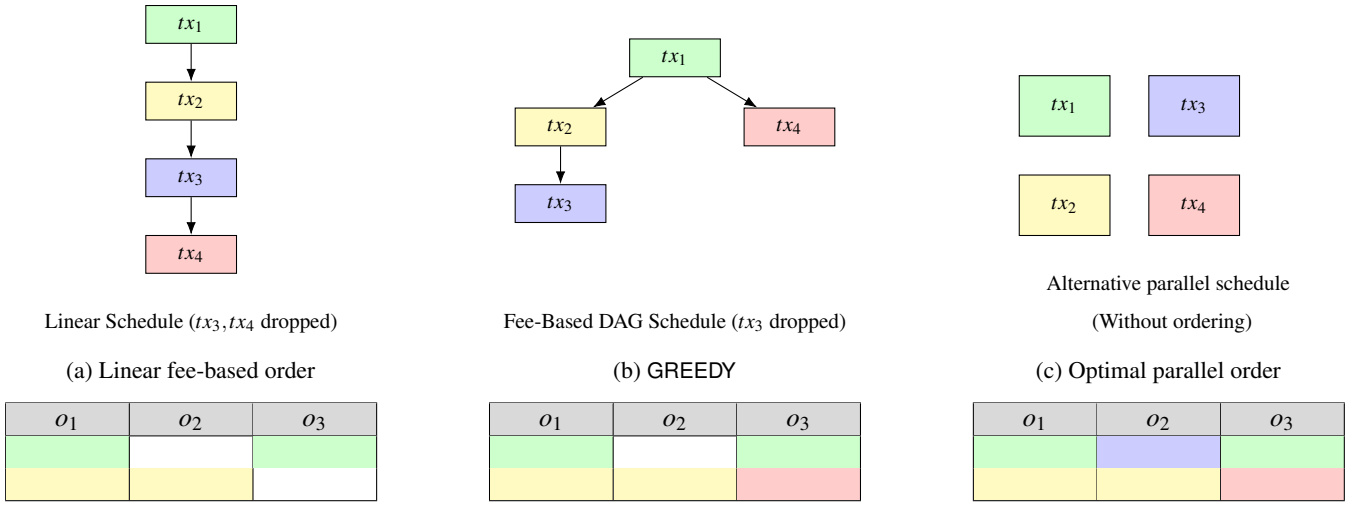


Figure 1: Comparison of scheduling strategies and their object usage.

revenue of the greedy schedule, and all parallel transactions without scheduling tx. Thus, $R_{\mathsf{OPT}} \leq R_{\mathsf{GREEDY}} + T_{\mathsf{max}} \cdot 1 + (G \cdot 1) \cdot (\text{number of parallel transactions - } 1) < G \cdot$ number of parallel transactions. The number of parallel transactions can be at most the number of cores, and also at most the number of objects (otherwise, by pigeon hole principle, two transactions in parallel would have to use the same object). Thus, number of parallel transactions $= \min(n, n_O)$, and,

$$\alpha(\mathsf{GREEDY}) = \frac{R_{\mathsf{GREEDY}}}{R_{\mathsf{OPT}}} > \frac{G - T_{\mathsf{max}}}{G \cdot \min(n, n_O)}.$$

Next, we show that this revenue lower bound is achievable. Consider a set of transactions $\mathcal{T}_s = \{tx_i\}$ such that $tx_i \simeq (t_i, \{o_1\})$, gas price is $1+\varepsilon$ for each of them and $\sum_i t_i = G - T_{\mathsf{max}}$. Next, consider a transaction $tx \simeq (\varepsilon, O)$, gas price is 1. Finally, consider another set of transactions $\mathcal{T}_n = \{tx_j\}$ such that $\forall o_j \in O : tx_j \simeq (T_{\mathsf{max}}, \{o_j\})$ with a gas price of $1-\varepsilon$ and consider enough instances of each tx_j to fill the computation limits of the block. If the partial order follows GREEDY, then only transactions in $\mathcal{T}_s$ and tx would be scheduled, leading to a revenue of:

$$R_{\mathsf{GREEDY}}(\mathcal{T}_s, tx, \mathcal{T}_n) = \overbrace{(1+\varepsilon) \cdot (G - T_{\mathsf{max}})}^{\mathcal{T}_s} + \overbrace{1 \cdot \varepsilon}^{tx}$$

Considering T_{max} to be a factor of G, the optimal revenue in this case would be:

$$R_{\mathsf{OPT}}(\mathcal{T}_s, tx, \mathcal{T}_n) = \overbrace{(1+\varepsilon) \cdot (G - T_{\mathsf{max}})}^{\mathcal{T}_s} + \overbrace{(1+\varepsilon) \cdot T_{\mathsf{max}}}^{tx_1 \in \mathcal{T}_n} + \overbrace{(1-\varepsilon) \cdot G \cdot (\min(n, n_O) - 1)}^{\forall tx_j \in \mathcal{T}_n, j \neq 1}$$

Thus, $\lim_{\varepsilon \to 0^+} \alpha(\mathsf{GREEDY}) = \frac{G - T_{\mathsf{max}}}{G \cdot (\min(n, n_O)}$. □

Randomized Order. Many systems implement RANDOM: e.g., Aptos performs a random shuffle (with anti-same-sender heuristics) after consensus [3]; in Monad [26], consensus fixes order, however, no ordering policy is specified to the consensus. This would yield a distribution of transactions that we treat as random for analysis. Since GREEDY is a valid instantiation of a random ordering, the worst case for RANDOM cannot be better:

Lemma 6. *The lower bound for* $\alpha(\mathit{RANDOM})$ *is at least as bad as the lower bound for* $\alpha(\mathit{GREEDY})$.

Proof. Any GREEDY output is feasible for RANDOM, so RANDOM's worst case is no better than GREEDY's. □

Remark on Optimistic Execution (Monad). In optimistic execution, transactions start immediately after consensus without a precomputed schedule; conflicts trigger re-execution. If a fee-ordered consensus sequence places a large set $\mathcal{T}_s$ of mutually conflicting transactions first, many cores can be kept busy executing work that later aborts, deferring useful parallelism. When early transactions in $\mathcal{T}_s$ use ε compute while others consume near T_{max}, the first successful commit delays re-execution by $\approx T_{\mathsf{max}}$, potentially yielding revenue strictly below a simple RANDOM baseline despite high core utilization (wasted on aborts). This phenomenon does not alter the proofs above; it illustrates that *execution policy* can further degrade realized revenue beyond ordering alone.