Applications of Cost-sharing mechanism on state-rental

Bruno Mazorra
brunomazorra@gmail.com

June 5, 2024

Abstract

Since L2s proceed much quicker than Ethereum, their state also grows quicker. This raises a concern that the "traditional" approach to blockchain state storage, i.e., buy storage once and keep it forever, cannot work anymore. The quick state growth would require the full nodes to constantly improve their storage capabilities, which may lead to undesirable network centralization. One of the ideas to mitigate this problem is "state rental". The solution assumes that whatever is written to the blockchain (by smart contracts/accounts) doesn't need to be stored permanently, but can be removed after some time of inactivity, i.e. the state space is rather rented than sold to a smart contract. In this work, we will formalize the problem and provide a state rental mechanism with desirable properties.

Keywords: Mechanism design, Blockchain, State rent, Cost sharing

1 Introduction

In economic terms, the fundamental problem of unrestricted state growth in a blockchain like Ethereum, without a mechanism like state rent, can be understood through the lens of externalities. An externality occurs when a decision affects a third party not directly involved in the decision-making process. An example of a negative externality in a common context is air pollution from factories. When a factory produces goods, it may also emit pollutants into the air. This pollution can affect the health and environment of people living in nearby areas who are not involved in the factory's operations or decision-making processes. These individuals bear the cost of the factory's pollution (such as health problems or reduced quality of life) without having any direct involvement in the factory's production activities. This impact on third parties, not directly involved in the economic activity of the factory, exemplifies a negative externality. In the case of blockchain state growth, there are two key externalities to consider.
The first major negative externality of state growth is the increased operational costs for node operators. As the Ethereum blockchain accumulates more smart contracts and accounts, its overall state size expands. This growth leads to escalating costs for those maintaining the nodes, including higher expenses for hardware, bandwidth, and energy required to store, process, and validate the state. However, the creators and users of these contracts and accounts do not directly bear these operational costs.
Another critical issue is the barriers to entry and centralization risks that a larger state size brings. Operating a node with a vast state requires advanced and often expensive hardware. This necessity raises the entry threshold for new node operators, potentially decreasing the number of full nodes. Such a reduction could tilt the network towards centralization, threatening the blockchain's security and resilience by making it more susceptible to censorship and attacks. Furthermore, an expanding state can degrade the overall network performance. A bloated blockchain might experience slower transaction processing times and present significant challenges in syncing new nodes to the network. These performance issues are not limited to those contributing to the state size but rather affect all users of the blockchain network. This broad

impact further underscores the shared burden of blockchain state growth, highlighting the need for mechanisms like state rent to manage and mitigate these externalities.

The current design of blockchains such as Ethereum, which lacks a state rent system, exhibits a significant misalignment between the actions of smart contract creators/users and the broader network. This misalignment stems from the fact that creators and users benefit from the blockchain's resources without bearing the full cost of their consumption, a situation reminiscent of the "tragedy of the commons". Such a setup leads to overuse and inefficient allocation of blockchain resources, as there is no direct cost for storage space. Presently, users pay a one-time gas fee to store data or change the state on the Ethereum blockchain, but this does not include ongoing costs for maintaining that data on the network. Consequently, users who add to the state size are not continually taxed for the externalities their actions impose on the network.

Implementing a state rent protocol could be a potential solution to these issues. A state rent system would internalize the external costs by requiring users to periodically pay for their use of storage space. This approach would create a direct financial incentive to use storage efficiently. Additionally, by attaching a continuous cost to blockchain storage, users would be encouraged to minimize unnecessary data storage and to clean up obsolete or unused data. This would lead to a more sustainable management of the state size. From an economic perspective, state rent can lead to a more efficient allocation of resources within the Ethereum ecosystem, aligning the private costs of individuals or entities with the social costs incurred by the network. This realignment could significantly improve the overall functionality and sustainability of blockchain systems like Ethereum.

However, the state rent presents several challenges. One of the primary challenges is the free-ride problem, where users or entities that utilize a contract benefit from the storage without necessarily contributing to its cost. Take, for example, a token contract used by thousands of users. The question arises: who bears the cost of storing the smart contract? If the cost falls on the smart contract creator, they could become vulnerable to griefing attacks, as detailed in [Akh23]. This situation creates a potential imbalance where the burden of maintaining the contract disproportionately affects the creator.

Another significant challenge is the issue of smart contract dependencies. Many smart contracts are interlinked, relying on one another for data and functionality. If a contract is evicted due to non-payment of state rent, it could break or disrupt the functionality of dependent contracts, potentially triggering a cascade of issues across the network. This interconnectedness adds a layer of complexity to managing and implementing state rent policies. The eviction of one contract can have unforeseen and widespread consequences on others, making it difficult to predict and manage the overall impact on the ecosystem.

Furthermore, implementing a state rent system could notably affect the user experience. For users, particularly those less technically sophisticated, the need to continually monitor and pay rent for contract storage could introduce additional complexity and stress. This is especially concerning if there is a risk of eviction for non-payment. Such a system might make the Ethereum platform less user-friendly and accessible, potentially deterring casual or new users who might find these additional requirements intimidating.

Finally, the aspect of re-instantiating evicted contracts presents another challenge. This process requires careful design to ensure that contracts can be restored effectively without loss of data or functionality, all while maintaining network integrity and security. The re-instantiation process needs to be seamless and user-friendly to avoid creating barriers to entry or additional burdens for users. Overall, these challenges highlight the need for a nuanced and well-thought-out approach to implementing state rent in Ethereum or similar blockchain platforms.

2 Summary of the mechanism

The Constant Function Market Makers (CFMM) mechanism for state rental in Ethereum-like blockchains is characterized by the following features:

1. **Token-Based Mechanism:** The solution involves two native tokens: T for block space and sT for storage, to manage state rent efficiently.

2. **Loss Function for Storage Cost:** The mechanism integrates a loss function representing the social cost of storage usage, which helps to internalize the external costs of storage on the Ethereum network. The users pay their marginal contribution in the social cost generated.

3. **Rent Field and Eviction in Accounts:** Each account includes an additional field for rent and a functionality for eviction. The rent field contains two parameters, how much tokens does it have locked for paying the rent and when was the last time the account payed the rent. The other functionality allows the owner of the account or other accounts under some constraints to remove the account from the database. The users pay 1) every time they use their account or 2) explicitly by adding storage tokens. The cost of the storage is path-dependent and so they have to pay in terms of the average demand between the rent payment date and the last rent payment date.

4. **Cost Sharing for Smart Contracts:** To address the free-rider problem in smart contracts, the system tracks sT tokens stored in contracts for each user, proposing a fair and efficient method for distributing storage costs among smart contract users. In other words, agents have to pay their marginal contribution on the cost of the smart contract.

5. **Re-instantiating Evicted Contracts:** The system includes a process for restoring smart contract states post-eviction, ensuring the continuity and integrity of valuable contracts and accounts.

3 A CFMM Solution for state rental applied accounts

We propose a CFMM type of solution for state rental by creating a token and an automated market maker that is "pegged" to the cost of storage given the storage supply and demand. To do so, the chain will have two native tokens T (paying for block space) and sT (paying for storage). Each account will have an extra field called `rent` and an extra functionality `eviction`, that once executed erases the account from the database. The field `rent` contains two parameters `rent.balance` and `rent.lastPayment`. The function `eviction` can be called in two different ways. By the owner of the account by selling the storage to the CFMM or by any agent when the `rent.Balance` < 0, obtaining some reward for doing so (this reward should depend on the demand of storage, the demand of blockspace, the inflation of the native token, etc. This reward is out of the scope of this note).
Similar to [Dia+23b], we define a *loss function* of the social cost induced by the storage $\ell : \mathbb{R}_+ \to \mathbb{R}_+ \cup \{+\infty\}$, which maps the total amount of storage used x of the blockchain to the overall social cost of the network $\ell(x)$ for a unit of time. A natural function is $\ell(x) = C_1 e^{(C_2 x)} - 1$ with $C_1, C_2 \geq 0$. If the total amount of storage used is x for t units of time, the social cost is $t\ell(x)$. This model assumes that this map is convex, $\ell(0) = 0$ and twice differentiable. Since the map is positive, $\ell(0) = 0$ and convex, we have that is monotone non-decreasing. Moreover, we assume that $L_{\min} := \operatorname{arginf}\{\ell(L) : L \geq 0\} < \infty$. This can be interpreted as there is a finite amount of storage that, if being used, breaks the fundamental guarantees of the underlying Blockchain such as Liveness, making the chain unusable for everyone (colloquially nominated as "brick wall" by Péter Szilágyi [Con22]).

Now, suppose that the blockchain currently has a total storage of R, and an agent (user) wants to create an account or a Smart contract that uses x units of storage. This operation will increase the total storage from R to $R+x$, increasing the social cost from $\ell(R)$ to $\ell(R+x)$. To bear the social costs, the agent should, at least, pay to the network, its externality $\ell(R+x)-\ell(R) =: f_2(x)$. On the other hand, if an agent liberates x units of storage, they should be paid the positive externality generated $\ell(R)-\ell(R-x)$. In terms of CFMM literature [Dia+23a], we can define a trading function φ that captures the "forward exchange function" $f_1 : y \mapsto \ell^{-1}(y+\ell(R))$ (if an agent pays y T obtains $f_1(y)$ sT) and defines a CFMM. Let $\varphi(x_1, x_2) = \ell(L_{min} - x_1) - x_2$ be the trading function of a CFMM with tokens sT (storage of native token T) and T and initial reserves (L_{min}, C), with C some amount of tokens T. These tokens do not necessarily have to exist and can be mint and burn at will by interacting with the CFMM[1]. Is easy to show that this trading function has as forward exchange function the marginal social cost per unit of storage.
Now let $g : [t_1, t_2] \to \mathbb{R}_{\geq 0}$ be a measurable function such that $g(t)$ represents the total storage used at time t. Now, lets assume that an agent consumes x units of storage from t_1 to t_2. Moreover, assume that $x << g(t)$ for all $t \in [t_1, t_2]$ then, the externality generated is:

$$\begin{aligned} y(t) &= \int_{t_1}^{t_2} \ell(g(z)) - \ell(g(z) - x)dz \\ &\approx \int_{t_1}^{t_2} x\ell'(g(z))dz \\ &= x\int_{t_1}^{t_2} \ell'(g(z))dz \end{aligned}$$

And so, the agent should pay at least $y(t)$. Observe that $\ell'(g(z))$ is the price of buying an infinitesimal amount of storage when the network is using $g(z)$ units of storage. And so, $\int_0^t \ell'(g(z))dz$ is the average price of storage on the interval $[t_1, t_2]$.
Every time an account `Addr` does an operation (like a transfer), it updates `rent` as follows:

$$\begin{aligned} \texttt{rent.balance} &\leftarrow \texttt{rent.balance} \ - \int_{t_1}^{t_2} \ell(g(z)) - \ell(g(z) - x)dz \\ \texttt{rent.lastPayment} &\leftarrow \texttt{currentBlock} \end{aligned}$$

where $t_1 =$ `Addr.rent.lastPayment` before the update and $t_2 =$ `currentBlock`. The `rent` variable can also be updated by any user if $\int_{t_1}^{t_2} \ell(g(z)) - \ell(g(z) - x)dz >$ `rent.balance`, erasing the account from the database.
Optionally, we can also think of interacting with the address in a way such that the user is expressing their preference to maintain the address. And so, we can add to the update part of the base fee to the balance. However, we can not add it completly since then that will change the economic EIP1559 guarantees, since the user could pay all the base fee. So we define the following update

$$\texttt{rent.balance} \leftarrow \texttt{rent.balance} \ - \min\left\{\int_{t_1}^{t_2} \ell(g(z)) - \ell(g(z) - x)dz + f_1(\texttt{baseFee}), 0\right\},$$

where `baseFee` is the base fee to the transaction. This gives a better user experience for the users, where they are already paying the storage of the address by using it.
With this proposal, we successfully sketch a market for storage, where each agent pay the marginal externality it generates. However, do all payments cover the social cost generated?

[1]Observe that if currently, there are R sT tokens holder by users, then the CFMM holds $L_{min} - R$ tokens and all the users hold R.

Mathematically this translates to the following. Assume n agents use storage $x_1, ..., x_n$ over a unit of time. Then, the agents pay the social cost generated to the network if and only if

$$\sum_{i=1}^{n}[\ell(\overline{x}) - \ell(x_{-i})] \geq \ell(\overline{x}) \quad \text{(self-bounding)}$$

where $\overline{x} = \sum_{j=1}^{n} x_i$ and $x_{-i} = \overline{x} - x_i$. This follows, from the convexity of ℓ and that $\ell(0) = 0$. Let's prove it. First, observe that $\ell(\overline{x}) - \ell(x_{-i}) = \ell(x_1 + ... + x_i + a) - \ell(x_1 + ... + x_{i-1} + a)$ with $a = x_{i+1} + ... + x_n$. By convexity, we have that $\ell(x_1 + ... + x_i + a) - \ell(x_1 + ... + x_{i-1} + a) \geq \ell(x_1 + ... + x_i) - \ell(x_1 + ... + x_{i-1})$, when $i \geq 2$ and $\ell(x_1 + a) - \ell(a) \geq \ell(x_1) - \ell(0) = \ell(x_1)$. Therefore,

$$\begin{aligned}\sum_{i=1}^{n}[\ell(\overline{x}) - \ell(x_{-i})] &\geq \ell(x_1) + \sum_{i=2}^{n} \ell(x_1 + ... + x_i) - \ell(x_1 + ... + x_{i-1}) \\ &= \ell(\overline{x})\end{aligned}$$

Now, if the storage demand curve of each player i change over time $x_i(t)$, we have the same, result, that is the sum of payments cover the social cost. Similar as before, let $\overline{x}(t) = \sum_{i=1}^{n} x_i(t)$ and $x_{-i}(t) = \overline{x}(t) - x_i(t)$.

Proposition 1. The sum of payments on $[t_1, t_2]$ cover the social cost induced by the agents on time $[t_1, t_2]$. More formally

$$P(t) = \sum_{i=1}^{n} \int_{t_1}^{t_2} \ell(\overline{x}(t)) - \ell(x_{-i}(t))dt \geq \int_{t_1}^{t_2} \ell(\overline{x}(t)).$$

This market structure incentivizes users to not misuse the storage of the chain and also to free space when the market values more the storage than This is true under mild conditions on users' demands and the CFMM curve, see [FDPR12; FPW23]. Moreover, inactive addresses are eventually evicted.

4 Cost sharing the storage costs of Smart contracts

In blockchain platforms like Ethereum, which are Turing-complete, managing the "state rent" for smart contracts is a particularly unique challenge, resembling the management of a public excludable good. While it's relatively straightforward to handle state rent for individual accounts, the situation becomes more complex with smart contracts due to their nature as shared resources. These smart contracts often offer public services or goods. This leads to what's known as the "free-rider problem", where some users might benefit from the contract without contributing to its maintenance costs. Contrary to individual accounts, where the costs are solely the responsibility of the account holder, smart contracts serve a multitude of users. From an economic perspective, accounts can be considered private goods. In contrast, contracts for decentralized applications (dApps) like Uniswap pools can be interpreted as public excludable goods. This scenario calls for a fair and efficient model of cost-sharing, ensuring that all users benefiting from a smart contract also share its costs.
In this context, the state rent protocol will introduce a new mapping `balanceST` for each smart contract that keeps track of the sT tokens stored in the smart contract for each agent. Periodically, we update the `balanceST` for each smart contract with the following update rule.

$$\begin{aligned}&\texttt{balanceST}[\texttt{Add}] \leftarrow \texttt{balanceST}[\texttt{Add}] - p[\texttt{ADD}, \texttt{currentBlock}, \texttt{lastPayment}], \\ &\quad \texttt{lastPayment} \leftarrow \texttt{currentBlock}.\end{aligned}$$

Observe that the payment function is not defined. Informally, the payment function p should depend on the address contribution on the cost of storage and the total cost of the Smart contract.
Let's take a key example: An ERC20 token. Fundamentally, the storage of an ERC20 token, consists on the bytecode of the contract, the balances of all the users, and the total supply. Suppose that bytecode and the total supply occupy a storage of y and the storage per user is x, then the total storage is $S([i]) = y + ix$ where i is the number of agents that have a strict positive balance of tokens. Then, the total amount to be paid in a time frame is $L([i], R) = \ell(R + S([i])) - \ell(R)$. To split the costs, we can use the Shapley value mechanism or more generally the hybrid mechanism, proposed in [Dob+08]. Agents have to lock sT tokens to the smart contract. Each smart contract has to set an array of balances of slots, if the agent does not have enough to pay, erase its part of the memory, in the case of the ERC20 the balance. Now, if n players have strictly positive balance, then the total amount to be paid is:

$$C([n], t) = \frac{1}{n} \int_{t_1}^{t_2} L([n], x(t))dt$$

and the amount that each player paid is

$$P(t) = \frac{C([n], t)}{n}$$

from the tokens sT stored in the smart contract. If an agent does not have enough tokens locked in the contract, then the SLOT of memory of its balances is evicted. Similarly, if no subsets of agents can cover the cost of the contract, then the contract is evicted. More generally, let $C : 2^{[n]} \times \mathbb{R}_+^2 \times \mathcal{L}_2(\mathbb{R}_+, \mathbb{R}_+) \to \mathbb{R}_+$ (monotone) be the *social cost set map*. More formally, given two timestamps $t_1, t_2 \in \mathbb{R}_+$ and a measurable demand curve x of storage without taking into account the smart contract. We define C as:

$$C(M, t_1, t_2, x) = \int_{t_1}^{t_2} \ell(x(t) + S[M]) - \ell(x(t))dt$$

Then we define the following mechanism for sharing the storage costs of smart contracts.

Mechanism

Break the time interval $[a, b]$ in subintervals $t_1, t_2, ..., t_k$ with $a = t_1$ and $b = t_k$. For example epochs.
Initialize $N = [n]$. For $i = 1, ..., k - 1$:

1. Let $b_i = \texttt{balanceST[Add]}$.
2. Let $S^* = \arg\max_{S \subseteq N} \left[\sum_{i \in S} b_i - C(S, t_i, t_{i+1}, x)\right]$
3. Initialize $S = S^\star$.
4. If $b_i \geq C(S', t_i, t_{i+1}, x)/|S'|$ for every $i \in S$, then halt with winners S.
5. Let $i' \in S$ be a player with $b_{i'} < C(S', t_i, t_{i+1}, x)/|S'|$.
6. Set $S = S' \setminus \{i'\}$ and return to Step 4.
7. Charge each winner $i \in S$ a payment p_i equal to the minimum bid at which i would continue to win (holding b_{-i} fixed). For every player $i \in [n] \setminus S$, evict its storage from the Smart contract. Update, $\texttt{balanceST[Add]} \leftarrow \texttt{balanceST[Add]} - p_i$ and $\texttt{lastPayment} \leftarrow t_{i+1}$.

Note that when the cost function is submodular, the previous mechanism coincides with the Shapley value mechanism.
This mechanism exhibits essential properties that are foundational to its design. Firstly, it upholds the principle of *individual rationality*, ensuring that participation is advantageous for all agents involved. Secondly, it embodies *truthfulness*, a property that incentivizes agents to reveal their true valuations to the mechanism (under private valuations), i.e. the agents lock the amount of tokens they are willing to pay for using the contract. Lastly, the *no-deficit* property guarantees that the mechanism operates without incurring losses to the other users of the network (out pay the social cost given by ℓ. For a comprehensive understanding of these properties, refer to [Dob+08].
At its core, this mechanism addresses the complexities that arise when agents possess private valuations over shared resources, such as storage in a smart contract environment. It is particularly effective in a scenario where each agent controls only a single identity, thereby ensuring *strategy-proofness*. This aspect is crucial as it mitigates the "free-ride" problem, ensuring that all agents contribute equitably to the social cost of maintaining the smart contract's storage.
However, a notable limitation of this mechanism is its computational feasibility. As it stands, the mechanism does not guarantee computability within polynomial time. This issue can be addressed by adapting a similar approach proposed [GS19], which offers a more computationally viable solution.
Despite these strengths, the mechanism is not inherently Sybil-Proof, meaning it is susceptible to manipulation when agents create multiple identities. This vulnerability is documented in [Maz23]. Nevertheless, it is important to note that this susceptibility does not significantly impact the overall welfare under specific conditions, as also detailed in [Maz23]. Moreover, this mechanism is strong against griefing attacks [Akh23], since the attacker must cover the cost they are inducing to the network. We will discuss all these properties in future work.

5 Re-instantiating

Note: In the following section, we do not provide original work, for more details see [Akh23; Eth19], and we left the problems of incentives for future work.
Re-instantiating state in Ethereum's state rental model is a mechanism that allows for the revival of smart contract states or accounts after eviction due to inactivity or failure to pay rent. This feature is crucial for maintaining a balanced ecosystem where contracts (or accounts) can be restored to continue operations. It optimizes blockchain efficiency by clearing inactive contracts (or accounts) while providing a recovery option for the owners, fostering a more dynamic blockchain environment.
The solution for re-instantiating state involves several key steps:

1. **Storing Contract Snapshots**: When a contract is evicted, its state snapshot is stored, including bytecode, storage state, and associated data. This snapshot can be stored on a specialized blockchain section.

2. **Hash and Identifiers**: A unique identifier or hash is left on the blockchain upon eviction, acting as a reference to the stored snapshot.

3. **RESTORETO Opcode**: A new opcode in the EVM, `RESTORETO` [Eth19], is used for re-instantiating. It takes parameters like the identifier or hash stump and the restoration address.

4. **Verification and Validation**: Before restoration, the snapshot undergoes a verification process to ensure integrity and appropriate permissions.

5. **State Re-Injection and Re-activation**: Verified snapshots are re-injected into the blockchain, restoring the contract's bytecode, storage, nonce, and balance.
6. **Cost and Resource Management**: The cost associated with re-instantiating a contract is defined, considering resources for storage and computation.

An alternative approach is for users to store contract snapshots locally and provide them when re-instantiating:

1. **Snapshot Creation and Storage**: Users manually create and store snapshots of their contract's state locally.
2. **Hash Generation**: A cryptographic hash of the snapshot data is generated for later verification.
3. **Verification and Upload**: Users verify the snapshot's integrity and upload it to the blockchain for contract deployment.
4. **Re-establishing Connections**: Post-restoration, users must manage dependencies and external connections.

This local approach offers control and autonomy but comes with challenges in data integrity, security, and complexity. Users bear the responsibility for the security and integrity of their data, and ensuring blockchain consistency can be challenging.
In conclusion, re-instantiating state in Ethereum's state rental model, whether through on-chain mechanisms or local snapshot storage by users, presents a viable solution for managing contract continuity. It balances blockchain efficiency with flexibility for contract owners, contributing to a resilient Ethereum ecosystem.

6 Conclusion

We overviewed the state rental concept in Ethereum blockchain. We explored the economic implications of unrestricted state growth and proposed solutions to mitigate these challenges. The main positions discussed in the note include:

- **CFMM Solution for State Rental Applied Accounts**: This involves a token-based mechanism, with two native tokens (T for block space and sT for storage) to manage state rent. Each account would have an additional field for rent and a functionality for eviction. The solution incorporates a loss function representing the social cost of storage usage, aiming to internalize the external costs of storage on the Ethereum network.
- **Cost Sharing for Smart Contracts**: We delved into the unique challenges of managing state rent for smart contracts, highlighting the free-rider problem. We suggested a model where the state rent protocol tracks sT tokens stored in smart contracts for each agent, proposing a mechanism to fairly distribute storage costs among users of a smart contract.

In conclusion, we presented a comprehensive framework for implementing state rent in blockchain environments like Ethereum. It addresses the economic externalities of state growth and proposes mechanisms to efficiently manage and mitigate these challenges. The solutions aim to balance blockchain efficiency with the flexibility and sustainability of contract and account management, ensuring the adaptability of the blockchain ecosystem in the face of rapid state growth. Nonetheless, future research should focus on addressing several key areas. These include developing strategies for managing evictions in the context of smart contract dependencies, devising a method to accurately compute the function C for any given contract, formally verifying the properties discussed throughout this paper, and exploring cybersecurity concerns related to eviction and re-instantiation processes.

References

[Dob+08] Shahar Dobzinski et al. "Is Shapley cost sharing optimal?" In: *Algorithmic Game Theory: First International Symposium, SAGT 2008, Paderborn, Germany, April 30-May 2, 2008. Proceedings 1*. Springer. 2008, pp. 327–336.

[FDPR12] Rafael Frongillo, Nicholás Della Penna, and Mark D Reid. "Interpreting prediction markets: a stochastic approach". In: *Advances in Neural Information Processing Systems* 25 (2012).

[Eth19] Ethereum Magicians. *Discussion about Storage Rent, Eviction, Archive Nodes, and Incentives*. `https://ethereum-magicians.org/t/discussion-about-storage-rent-eviction-archive-nodes-and-incentives/2352`. Accessed: 2023. 2019.

[GS19] Konstantinos Georgiou and Chaitanya Swamy. "Black-box reductions for cost-sharing mechanism design". In: *Games and Economic Behavior* 113 (2019), pp. 17–37.

[Con22] Dev Connect. *Ethereum in numbers: Where TPS meets physics*. `https://www.youtube.com/watch?v=Cmuz_Xn_YJw`. [Online; accessed 10-December-2023]. 2022.

[Akh23] Alexey Akhunov. *State Rent Proposal*. `https://github.com/ledgerwatch/eth_state/tree/master`. 2023.

[Dia+23a] Theo Diamandis et al. "An Efficient Algorithm for Optimal Routing Through Constant Function Market Makers". In: *arXiv preprint arXiv:2302.04938* (2023).

[Dia+23b] Theo Diamandis et al. "Designing Multidimensional Blockchain Fee Markets". In: *5th Conference on Advances in Financial Technologies (AFT 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023.

[FPW23] Rafael Frongillo, Maneesha Papireddygari, and Bo Waggoner. "An Axiomatic Characterization of CFMMs and Equivalence to Prediction Markets". In: *arXiv preprint arXiv:2302.00196* (2023).

[Maz23] Bruno Mazorra. *On the optimality of Shapley mechanism under Sybil strategies*. `https://github.com/BrunoMazorra/WIP-papers`. 2023.