Introduction

In this post, we introduce MEV taxes, a mechanism that arbitrary applications can use to capture their own MEV.

This mechanism could be used today on OP Stack L2s like OP Mainnet, Base, and Blast, because the block proposers on those chains follow a set of rules we call competitive priority ordering

.

To implement a MEV tax on one of these chains, a smart contract charges a fee that is a function of the priority fee of the transaction. We show that if an application charges searchers a MEV tax of (say) $99 for every $1 of priority fee, it can capture 99% of the competitive MEV for that transaction.

MEV taxes are a simple technique that opens up a vast design space. You can think of them as allowing any application on the chain to run its own custom MEV auction, without needing any offchain infrastructure of its own, just by hooking into a single shared auction run by the block proposer.

We show how MEV taxes could be used to solve three major problems in MEV research:

- Decentralized exchange (DEX) routers that optimize the price received by the swapper
- Automated market makers (AMMs) that minimize the loss-vs-rebalancing (LVR) experienced by liquidity providers
- Wallets that let their users capture any “backrunning” MEV created by their transactions

Decentralized exchange (DEX) routers that optimize the price received by the swapper

Automated market makers (AMMs) that minimize the loss-vs-rebalancing (LVR) experienced by liquidity providers

Wallets that let their users capture any “backrunning” MEV created by their transactions

But there’s a catch. MEV taxes only work if block proposers strictly follow the rules of competitive priority ordering, which include sorting transactions by priority fee without censoring, peeking at, or delaying any. If block proposers deviate from those rules, they can evade MEV taxes to capture the value for themselves. Today, therefore, MEV taxes depend on trusting L2 sequencers, and would likely not work at all on Ethereum L1, where block building is dominated by a competitive builder auction that maximizes revenue for the proposer.

Still, the power and flexibility of MEV taxes suggests that priority ordering may be the right choice for platforms that can provide it today. And the relative simplicity of competitive priority ordering suggests that there may be a viable way to enforce it in a decentralized way, without having to trust a single sequencer. We hope this post motivates further work on that problem.

Priority ordering

When someone sends a transaction on an Ethereum L1 or L2, they specify a priority fee, which they pay to the block proposer.[1]

You can imagine this is specified as priorityFeePerGas

, a number that is multiplied by the gas used in the transaction to get builderPriorityFee

—the total payment in ETH[2]

There is no rule in the Ethereum protocol that transactions in a block must be ordered greedily by descending priorityFeePerGas

. However, that is a popular way to build blocks—for example, it is the default algorithm used by sequencers ofOP Stack chains, as well as geth and reth. Not only does priority ordering let transactors efficiently express the urgency of their transactions, it also naturally channels certain kinds of MEV to the block proposer.

That happens because priority ordering turns competition for MEV into apriority gas auction. When there is an opportunity to profit from interacting with the chain, such as by arbitraging an AMM against a centralized exchange, searchers compete to claim that opportunity first. If the chain uses priority ordering to determine transaction inclusion and ordering, the searchers compete by setting high priority fees on their transactions.

In a competitive scenario where risk-free profits are competed down to zero, the winning searcher should end up paying the full amount of MEV in priority fees.[3]

So if there is 100 ETH in profit to be gained from interacting with a contract, the first transaction to claim it will set a priority fee of 100 ETH. (We discuss some caveats to this in the Limitations section).

MEV taxes

Suppose a smart contract wants to capture the MEV from any transaction that interacts with it. There is a vast library of research on different application-specific ways that smart contracts could try to capture their own MEV.

But in fact, we don’t necessarily have to know anything about the application. If we know that the block is being constructed through competitive priority ordering, then we have one universal signal for the amount of MEV in the transaction: the priority fee.

We propose that the smart contract can look at the priority fee of the transaction and charge its own fee as some increasing function of it. For example, the contract might require whoever calls it to transferapplicationPriorityFee = 99 * proposerPriorityFee

in ETH to the contract[4]

This new fee is paid by the searcher sending the transaction, so it affects the behavior of that searcher. If there is 100 MEV in an opportunity, the winning transaction will now only set a priority fee of 1 ETH, since that will result in a total payment of 100 ETH (1 ETH to the block proposer, and 99 ETH to the smart contract). Any higher priority fee would make the transaction unprofitable; any lower priority fee would result in losing the opportunity to a competitor who set a higher fee. This means the smart contract has captured 99% of the MEV in the transaction.

We call this extra fee imposed by the smart contract a MEV tax

. MEV taxes let an application hijack priority ordering for its own benefit, allowing it to recapture MEV for its users rather than leaking it to the block proposer.

If this fee increases sufficiently fast as a function of priorityFeePerGas

, then only a negligible amount of MEV will accrue to the proposer. Since priorityFeePerGas

is denominated in wei (one billionth of a billionth of one ETH), we have a lot of precision to work with. For example, as long as the MEV tax is sufficiently sensitive that a priorityFeePerGas

of 50,000 would result in a prohibitively high tax, then the total payment to the proposer would be less than $0.01[5].

However, there is an important caveat. As discussed in the Limitations section, MEV taxes only work if block proposers follow certain rules—what we call "competitive priority ordering"—rather than deviating from those rules in order to maximize their own revenue. Enforcing these rules in a trustless way is an open problem.

Single-application MEV capture

Here we sketch out how, on a chain that is guaranteed to use competitive priority ordering for block building, MEV taxes could be used to mitigate three important problems in MEV: letting DEX interfaces improve trade execution for swappers, letting AMMs reduce losses to arbitrage for their LPs, and letting wallets reduce MEV leakage for their users by selling the right to backrun the user.

DEX routers

In intent-based DEX routing protocols like UniswapX and 1inch Fusion, a user (Alice) signs an intent to swap, and searchers compete to route or fill that intent at the best possible price for Alice.

Current versions of UniswapX use two mechanisms to run that competition: a Dutch auction where Alice's limit price changes over time until a searcher fills it, and an initial offchain request-for-quote (RFQ) auction to set the starting price of that Dutch auction.

On a platform that guarantees competitive priority ordering, UniswapX could replace these with a single mechanism: a MEV tax. It could implement this by having the user sign an order that can be filled immediately by anyone, but with an execution price that is set as a function of the transaction's priority.

For example, if Alice has a UniswapX order to sell 1 ETH, she could define the execution price of the order to be minimumPrice + ($0.01 * priorityFeePerGas)

. minimumPrice

could be some fixed value that she expects to be significantly lower than the current price.

Searchers would compete to fill Alice's order by submitting transactions. Whichever transaction has the highest priority fee and doesn't revert would get to fill the order, which should guarantee that the swapper gets the best price that searchers can find. (Some exceptions to this are discussed in the Limitations section.)

If Alice's minimum price is $3,000 and the current price of ETH is $3,500, priorityFeePerGas

in the winning transaction would be about 50,000. (Observe that in a transaction that costs 200,000 gas, this will result in a payment of only about 10 billion wei—around $0.000035—to the block proposer.)

This has some potential benefits over the existing mechanisms used in UniswapX.

Orders that use MEV taxes could complete faster and at a better price than orders which use Dutch auctions. As discussed in this paper, onchain Dutch auctions leak some value to MEV due to price movements between blocks, and may take many blocks to complete. In contrast, orders that use MEV taxes could typically be completed in the next block while capturing the vast majority of their MEV.

Unlike an offchain RFQ, the auction to fill an order that uses MEV taxes would happen atomically with transaction execution onchain. This means that a winning bidder could be guaranteed that they are only committed to fill the order if their onchain transaction succeeds. That could make it easier for onchain liquidity like AMMs to compete with offchain liquidity, meaning UniswapX could serve as an even more effective router for multi-pool systems like Uniswap v4.

AMMs

Normally, AMMs leak value to arbitrageurs who trade against stale prices at the top of the block, as discussed in the loss-vs-rebalancing papers. We can use MEV taxes to have AMMs capture that MEV. To keep things simple, we'll discuss how this might work on an AMM without concentrated liquidity. (If you're interested in how this kind of problem could be solved with concentrated liquidity, Sorella will soon be publishing one solution.)

An AMM can capture MEV by charging an extra fee as a function of the priority fee on the transaction, allowing it to auction off the right to trade first in the block. There are many ways to calculate and denominate that fee. We'll discuss one arguably neutral one—denominating it in units of pool liquidity, sqrt(xy)

. The winning transaction would be the one that increases the pool's liquidity by the most.

When executing the first transaction on a pool in a block, instead of enforcing the condition x_end * y_end > x_start * y_start

, the pool could enforce the condition (with a

as some constant):

This formula would incentivize the arbitrage trader to trade to the true price, and after that trade, the midpoint price on the pool should be the true price.6

After that first transaction, trades could work like they do on Uniswap v2, with fixed swap fees. Uninformed transactions that want to trade on the pool without paying an extra MEV tax would set a low priority fee.

There are many other ways to implement MEV taxes on an AMM that would have different effects. For example, MEV taxes could be denominated in the input or output token of the swap, could affect the swap fee percentage applied by the pool, or could determine the minimum price of the user's trade. We think this is an interesting design space to explore.

Backrunning auctions

The above descriptions show how certain applications could be designed to avoid leaking MEV. However, what if a wallet wants to try to help its users capture the MEV they create from arbitrary transactions interacting with any

application, even ones that don't incorporate MEV taxes?

For example, when Alice makes a large transaction on an AMM, she sometimes create an arbitrage opportunity for "backrunners" to move the price back. This is normally leaked to MEV, rather than going to Alice.

MEV-Share and MEVBlocker are two protocols that allow users to capture MEV from their transactions, but they rely on a complex offchain auction system. The Orderflow Auction Design Space describes some other solutions.

MEV taxes, when combined with an intent-based smart contract wallet, could allow us to construct an alternative system to capture backrunning MEV for Alice. Suppose that instead of creating a transaction that trades on the AMM, Alice signs an intent that anyone can submit to Alice's smart contract wallet to cause it to take that action. Alice's smart contract wallet charges whoever submits that transaction a MEV tax, which is paid to Alice.

The searcher who submits Alice's intent will have the exclusive right to backrun her, since they can do so atomically in the same transaction. As a result, if searching is competitive, all of the profit from backrunning Alice should accrue to Alice through her MEV tax.

Note that this system may not necessarily protect users from attacks that involve frontrunning

user transactions, because a transaction that frontruns a user may be able to avoid paying a MEV tax to that user. This issue (and some possible mitigations for it) is discussed in greater detail in the Limitations section below. Nevertheless, this could at least be an improvement on systems that use public mempools without any mitigations.

Other use cases

In addition to these examples, other potential uses of MEV taxes could include almost anything that currently uses an offchain or Dutch auction, such as:

- Protocols for oracles to capture the oracle extractable value they create, like Oval
- Refinancing auctions in NFT-collateralized lending protocols like Blend

- Lending protocol liquidations that leak less value than Dutch auctions

Protocols for oracles to capture the oracle extractable value they create, likeOval

Refinancing auctions in NFT-collateralized lending protocols likeBlend

Lending protocol liquidations that leak less value than Dutch auctions

Cross-application MEV capture

The above solutions are designed to capture the MEV from interacting with a single application. But sometimes it may be possible for a searcher to capture even more value by interacting with multiple applications in the same transaction.

If only one of those applications has a MEV tax, then all the MEV from the transaction should go to the application with the MEV tax, regardless of how high or low that MEV tax is.

But what if a searcher's transaction interacts with two applications that use MEV taxes? For example, what if there is some MEV that can only be captured by filling one of the MEV-taxed UniswapX orders described above against a MEV-taxed AMM?

In that case, the relative amount of excess MEV captured by each application is determined by how those applications set their MEV taxes. If the value app_i

charges as a MEV tax is given by the function tax_i(priority)

, then the priority of the winning transaction can be determined by solving for priority in this equation:

(Technically, we could add a third term for priorityPerGas * gasUsed

to account for the priority fee paid to the block proposer, but we will ignore that since, as discussed in Appendix A, it will likely be negligible under normal conditions.)

In the simple case of MEV taxes that are linear in priorityPerGas

(so tax_1(priorityPerGas) = a_1 * priorityPerGas

), you can solve for the share of MEV received by each application:

When setting its own MEV tax, an application faces a tradeoff—higher taxes let it capture a greater share of cross-application MEV when it occurs, but mean it could miss out on some cross-application MEV if there are competing ways to extract it. For example, if there is an AMM that charges a MEV tax on every trade, then a MEV-tax UniswapX order might be more likely to be filled by a different AMM or an offchain filler.

In many cases, there may be an equilibrium in which two applications design their MEV taxes in order to share MEV in a way that maximizes each of their welfare. For example, a MEV-tax AMM would likely want to capture value from a single informed trader near the top of the block, but then would want to provide liquidity to other traders and applications (including ones that use MEV taxes) with a low fixed fee. In that case, the AMM is likely to set a relatively low MEV tax (say, $0.00001 * priorityFeePerGas

), so that the arbitrage transaction (if any) happens early in the block, and then charge no MEV tax on subsequent transactions in the block. Applications like UniswapX that want to interact with the AMM can set a much higher MEV tax (say $0.01 * priorityFeePerGas

), to ensure that their transactions are included after the pool is already arbitraged. With those relative taxes, the AMM would end up arbed first even if there was only $1 of MEV on it and $50,000 of MEV in a UniswapX order.

We think this is a broad design space worthy of future study.

Limitations

MEV taxes have some complications and drawbacks. We think each of these is an interesting area for future research.

Incentive incompatibility

MEV taxes are not incentive-compatible for a monopolistic block proposer. They only work if there is fair competition for transaction inclusion, which can only happen if the block proposer follows rules that we'll call "competitive priority ordering," rather than maximizing their own revenue. Informally and non-exhaustively, we suggest that these rules should include:

- Priority ordering. Transactions within a block must be ordered in descending order of priorityFeePerGas

.

- Censorship-resistance. If the block proposer receives a transaction t1 during the block, and the block is either not full or includes some transaction t2 such that t2.priorityFeePerGas < t1.priorityFeePerGas

, then the block must include transaction t1.

- Pre-transaction privacy. The block proposer must accept transactions through a private endpoint and must not share such transactions with anyone else before committing to the block, or use the content of those transactions as an input in constructing its own transactions.

- No last look. The block proposer must set a definite time blockTime

before which they accept transactions from anyone, and after which they do not accept transactions from anyone.

Priority ordering. Transactions within a block must be ordered in descending order of priorityFeePerGas

.

Censorship-resistance. If the block proposer receives a transaction t1 during the block, and the block is either not full or includes some transaction t2 such that t2.priorityFeePerGas < t1.priorityFeePerGas

, then the block must include transaction t1.

Pre-transaction privacy. The block proposer must accept transactions through a private endpoint and must not share such transactions with anyone else before committing to the block, or use the content of those transactions as an input in constructing its own transactions.

No last look. The block proposer must set a definite time blockTime

before which they accept transactions from anyone, and after which they do not accept transactions from anyone.

If one or more of these properties is violated, it may weaken the effectiveness of MEV taxes. A block proposer that violates censorship-resistance can avoid most MEV taxes by excluding competing transactions and submitting a zero-priority transaction that takes the opportunity for itself. A block proposer that violates pre-transaction privacy could steal MEV from other transactions or peek at their priority fees to know exactly how high it needs to set its own, while one that is able to submit transactions later than anyone else would have a free "last look" on whether to outbid others for an opportunity, either of which could create adverse selection problems that ultimately discourage competition.

Unfortunately, while the first property would be easy to enforce at the protocol layer, enforcing the other properties trustlessly is an open problem.

In the absence of enforcement at the protocol layer, a single sequencer who commits to these rules needs to be trusted not to deviate from them, and if proposers outsource block building to a competitive revenue-maximizing auction (such as Ethereum L1's MEV-Boost), blocks will likely not follow them.

These issues can be "solved" with a single trusted sequencer who commits to use competitive priority ordering for block building. They may also be solvable with a decentralized mechanism using some combination of consensus, cryptography, and/or trusted execution environments, such as Sorella's Angstrom, Flashbots's SUAVE, Leaderless Auctions, or Multiplicity.

Full blocks

One exception to the normal operation of MEV taxes happens when blocks are completely full. In that case, block proposers may have to leave out lower-priority transactions, rather than simply including them late in the block. Since transactions that interact with MEV-taxed applications are likely to have extremely low priority fees, those applications are likely to be crowded out by applications that don't use MEV taxes, or ones that have extremely low MEV taxes. However, in a chain that uses an EIP-1559-like mechanism to set a separate basefee, it should be relatively rare for blocks to be completely full. Additionally, given that some

transactions need to be delayed when blocks are full, delaying transactions that express lower urgency by setting higher MEV taxes may be a reasonable result.

Reverted transactions

MEV taxes effectively rely on single-block auctions in which every "bid" is a transaction. One downside of those auctions is that losing bids will generally result in reverted transactions being included onchain, paying some basefee and congesting the chain.

If a sequencer can exclude failed transactions entirely, that would alleviate this issue, though that can be difficult to implement even with a centralized sequencer. (It would also not strictly obey the censorship-resistance property described above, though that definition could be adjusted.) A more sophisticated sequencer may be able to optimize this process by allowing transactions to specify which contentious auctions they are participating in, giving the sequencer enough information to skip subsequent transactions that it knows would fail.

Leaking of user intents

MEV taxes only work if there is competition among searchers, which means the opportunity needs to be somewhat widely known. For applications like AMMs, where the opportunity is visible onchain, that should happen naturally. But for applications like intent-based routing or backrunning auctions, that means the application may need to share the user's intent with searchers.

In some cases, the temporary privacy lost from broadcasting the user's intent before it is fulfilled may leak value in a way that cannot be recaptured by a MEV tax.

For example, suppose Alice wants to purchase a low-liquidity token using the backrunning auction protocol described above. She publishes a signed intent for her smart contract wallet to purchase that token on an AMM, setting some slippage tolerance. Searchers could race to push the price of that token to her slippage tolerance in a high-priority transaction, without

filling the user's order. The winner, Bob, could then non-competitively fill Alice's intent by including and backrunning it in a low-priority transaction, thus sandwiching Alice's trade and giving her a worse price while evading her MEV tax. A similar issue could happen with purchases of NFTs.

Note that such an attack would be risky for Bob, since he would not be able to guarantee atomicity between buying the token and selling it to Alice. A naïve Bob could fall victim to a "sandwich ripping" trap in which Alice publishes an intent to purchase a worthless token from herself, causing Bob to purchase it in anticipation of sandwiching her trade, but Alice revokes her intent before Bob is able to complete the sandwich.

Applications may also be able to mitigate this by limiting the set of searchers with which they share intents and monitoring their behavior, as many existing orderflow auctions do.

It may also be possible to combine MEV taxes with privacy-aware builder features like envisioned in Flashbots's designs for SUAVE.

Finally, in cases where Alice decides that the costs of sharing her intent outweigh the benefit from competitive searching, she could construct a transaction herself and submit it directly into the block. As discussed above, an ideal implementation of competitive priority ordering would provide pre-transaction privacy from the block proposer.

Discussion and prior work

Priority gas auctions

. Some of the dynamics of priority ordering in decentralized blockchains were studied in theFlash Boys 2.0 paper, which coined the term "miner extractable value." That paper observed that Ethereum miners (when that network used proof-of-work) were already ordering transactions by priority, and that arbitrageurs were relying on that behavior to participate in "priority gas auctions" in which they bid for the right to be included first in a block, which led to much of the MEV from decentralized exchange arbitrage accruing to miners.

First come, first served

. Some attempts at MEV mitigation through transaction ordering rules, such asThemis or Arbitrum One's current sequencer,7

have focused on enforcing a different ordering rule, first come, first served

(sometimes called "fair ordering") where block proposers must order transactions in the order in which they see them.

Priority ordering takes a different approach—treating transactions that arrive within a given period equally, and ordering them instead by their declared priority.

First come, first served is difficult to enforce or even define in a real network environment with more than one validator. It also can result in wasteful latency races and spam even with a single trusted sequencer. Finally, MEV taxes may be able to eliminate certain kinds of MEV that first-come first-served ordering cannot, such as arbitrage profits from discontinuous "jumps" in asset prices. The potential advantages of priority ordering over first-come first-served ordering are somewhat related to the advantages of discrete-time over continuous-time exchanges discussed in Budish, Cramton, Shim (2015).

Meanwhile, while priority ordering seems to leak value to MEV by default, this post shows how applications can be designed to recapture it.

Fee sharing

. Blast, an Ethereum L2,shares a portion of both priority and base fees with the smart contracts accessed in a transaction.

MEV taxes allow something similar (at least for priority fees), but can be implemented at the application layer on any

chain that uses competitive priority ordering, without special support for fee sharing. They also allow applications to define their own taxes as custom functions of priority fee, providing more flexibility and potentially resulting in greater composeability of MEV-aware applications.

Trustless solutions

. This post focuses on the motivation for platforms to use competitive priority ordering—and ways to take advantage of platforms that do—rather than discussing how to trustlessly enforce it.

There has been significant prior discussion of each of the other properties required for competitive priority ordering. For example, in Fox, Pai, Resnick (2023), the authors discuss vulnerabilities in onchain auctions in the absence of censorship resistance, and describe a design for a censorship resistant auction using multiple concurrent proposers. However, they do not suggest a specific ordering for transactions.

There has been other research on constructing mechanisms for trust-minimized block building, including Flashbots's SUAVE, Sorella's Angstrom, Leaderless Auctions, Espresso and Offchain Labs' decentralized Timeboost, and mandated public transaction inclusion by Péter Szilági.

Conclusion

We hope this post encourages L2s to consider using priority ordering (as is supported by default in the OP Stack) and inspires applications to try out MEV taxes where supported.

We also hope it motivates further research into protocols for trust-minimized competitive priority ordering on both L1 and L2. If you're interested in collaborating on that problem, and are reading this before Thursday, June 6, you can still apply for a TLDR Fellowship to work on MEV-resistant L2 sequencers with Dan. Or feel free to just reach out todan@paradigm.xyz and dave@paradigm.xyz with ideas!

Footnotes

1. In this post, we use "proposer" to refer to the actor or process that determines what transactions are included in a particular block. On Ethereum L2s, this role is typically filled by a "sequencer." On Ethereum L1, it is filled by a specific Ethereum validator called a proposer, though often the proposer outsources the task of building the block to a competitive auction in which "relayers" and "builders" participate. The details of how these responsibilities are divided are out of scope of this post. ↩

2. The priority fee per gas is not actually specified explicitly in the transaction, but can be computed in it. The transaction specifies a gas price, but Ethereum also charges a base fee, which is taken out of the gas price and burned. The base fee should be ignored for purposes of MEV taxes, since it is not under the transactor's control. The priority fee per gas —the price for the part of the transaction fee that goes to the block proposer—can be computed in Solidity as priorityGasPrice = tx.gasprice - block.basefee

. ↩

1. Alternatively, we could simply define "MEV" to exclude any searcher profit and only refer to value that would go to the validator. ↩

2. Note that proposerPriorityFee

—equal to priorityFeePerGas

times the total gas used in the transaction—cannot actually be calculated during the contract, since there's no way to know how much gas the transaction will end up using. However, this generally won't matter, since all we need is an upper bound for it. To be safe, you could multiply priorityFeePerGas

by 30 million—the current maximum gas in an Ethereum block. Overestimating this value will simply mean that the MEV tax captures an even greater percentage of MEV. ↩

1. Assuming a transaction cannot be more than 30 million gas, a priorityFeePerGas

of 50,000 would result in a gas payment of 1500 gwei—about $0.006 at an ETH price of $4000↩

1. In the case where priorityFeePerGas is set so that the arbitrageur's profit is zero, the profit-maximizing arbitrage trade should correspond to the same trade on the function-maximizing AMM. Proving this is left as an exercise for the reader. ↩

2. Arbitrum has discussed replacing this with a form of priority ordering called Timeboost, but that has not been put into

production as of this writing. ↩

In this post, we use “proposer” to refer to the actor or process that determines what transactions are included in a particular block. On Ethereum L2s, this role is typically filled by a “sequencer.” On Ethereum L1, it is filled by a specific Ethereum validator called a proposer, though often the proposer outsources the task of building the block to a competitive auction in which “relayers” and “builders” participate. The details of how these responsibilities are divided are out of scope of this post. ↩

The priority fee per gas is not actually specified explicitly in the transaction, but can be computed in it. The transaction specifies a gas price, but Ethereum also charges a base fee, which is taken out of the gas price and burned. The base fee should be ignored for purposes of MEV taxes, since it is not under the transactor’s control. The priority fee per gas—the price for the part of the transaction fee that goes to the block proposer—can be computed in Solidity as priorityGasPrice = tx.gasprice - block.basefee

. ↩

Alternatively, we could simply define “MEV” to exclude any searcher profit and only refer to value that would go to the validator. ↩

Note that proposerPriorityFee

—equal to priorityFeePerGas

times the total gas used in the transaction—cannot actually be calculated during the contract, since there’s no way to know how much gas the transaction will end up using. However, this generally won’t matter, since all we need is an upper bound for it. To be safe, you could multiply priorityFeePerGas

by 30 million—the current maximum gas in an Ethereum block. Overestimating this value will simply mean that the MEV tax captures an even greater percentage of MEV. ↩

Assuming a transaction cannot be more than 30 million gas, a priorityFeePerGas

of 50,000 would result in a gas payment of 1500 gwei—about $0.006 at an ETH price of $4000.↩

In the case where priorityFeePerGas is set so that the arbitrageur’s profit is zero, the profit-maximizing arbitrage trade should correspond to the same trade on the function-maximizing AMM. Proving this is left as an exercise for the reader.↩

Arbitrum has discussed replacing this with a form of priority ordering called Timeboost, but that has not been put into production as of this writing. ↩