For your reference, the idea is inspired by this post.

Pay with MEV

In brief, DApps may “create” MEV on purpose, to pay transaction fees. More specifically, you send a predetermined amount of eth to block.coinbase instead of normal gasPrice * gasUsed

payment.

Usually, a transaction signed by an account is guaranteed to pay X eth per unit for at most Y gas. Block producers blindly choose transactions with the highest gas price and care nothing about what’s in the transaction. However, in the recent MEV era, some miners start to inspect transactions deeply and try to capture more profit than their face value(i.e., gas).

Flashbots’ mev-geth is doing so.

Some mining pools even include low or zero fee transaction that provides MEV.

(sometimes arbitrage bot, sometimes sandwich attack

victim)

And I’ll say that’s not a bad thing if we can utilize it to improve user experience.

Take Uniswap for example :

this is the original method of swapping ETH for ERC20 token:

function swapExactETHForTokens(uint amountOutMin, address[] calldata path, address to, uint deadline) external payable ensure(deadline) returns (uint[] memory amounts) { require(path[0] == WETH, 'UniswapV2Router: INVALID_PATH'); amounts = UniswapV2Library.getAmountsOut(factory, msg.value, path); require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT'); IWETH(WETH).deposit{value: amounts[0]}(); assert(IWETH(WETH).transfer(UniswapV2Library.pairFor(factory, path[0], path[1]), amounts[0])); _swap(amounts, path, to); }

Slightly modify this function by adding fee

into it, and you’ll get:

function swapExactETHForTokens(uint fee, uint amountOutMin, address[] calldata path, address to, uint deadline) external payable ensure(deadline) returns (uint[] memory amounts) { require(path[0] == WETH, 'UniswapV2Router: INVALID_PATH'); amounts = UniswapV2Library.getAmountsOut(factory, msg.value - fee, path); //reserve fee to pay miner require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT'); IWETH(WETH).deposit{value: amounts[0] - fee}(); assert(IWETH(WETH).transfer(UniswapV2Library.pairFor(factory, path[0], path[1]), amounts[0] - fee)); _swap(amounts, path, to); block.coinbase.call{value : fee }(""); //pay fee }

Since you already pay the fee inside

the transaction, gasPrice can be set to zero(or BASEFEE + 0 tip after EIP1559). By doing so, a miner that includes this transaction in a block would get paid only if the transaction gets executed successfully.

If this transaction gets reverted, no matter it’s because of slippage, insufficient token allowance, expiration, or what, the miner won’t get paid. The interest of user and miner now get aligned: miner including a failed swap costs user nothing and merely wastes the precious block space.

Pros:

- users no longer pay $80 for a failed swap
- flexibility of tx fee payment terms
- save block space by removing useless transactions

Cons:

- not every miner inspect MEV, it takes more time to get tx confirmed
- introduce extra complexity to measure blockchain congestion
- introduce extra complexity to adjust fee

- breaks the convention: MataMask warns user “gas price extremely low”

.

Suppose users are willing to pay more for a never-failing transaction, this should be a win-win situation.

Anyway, we can imagine DApps, especially those who create many reverted transactions, improve user experience by integrating this “pay with MEV” approach. In long term, the flexible fee-paying method may abstract out from the rigid tx fee system. While gas metering is written in consensus, how to pay for it can stay in between users and block producers.