zkSync Fee Mechanism

#

zkSync Fee Mechanism

This document will assume that you already know how gas & fees work on Ethereum.

On Ethereum, all the computational, as well as storage costs, are represented via one unit: gas. Each operation costs a certain amount of gas, which is generally constant (though it may change duringupgradesopen in new window).

zkSync as well as other L2s have the issue which does not allow to adopt the same model as the one for Ethereum so easily: the main reason is the requirement for publishing of the pubdata on Ethereum. This means that prices for L2 transactions will depend on the volatile L1 gas prices and can not be simply hardcoded.

#

gas_per_pubdata_limit

The transactions on zkSync depend on volatile L1 gas costs to publish the pubdata for batch, verify proofs, etc. For this reason, zkSync-specific EIP712 transactions contain thegas_per_pubdata_limit field in them, denoting the maximum price ingas that the operator __can charge from users for a single byte of pubdata.

For Ethereum transactions which do not contain this field, it is enforced that the operator will not use a value larger value than a certain constant.

#

Opcode Pricing

Zero-knowledge proof operations tend to have different complexity compared to standard CPU terms. For instance,keccak256 is optimized for CPU performance but costs more to prove in zero-knowledge systems due to mathematical constraints.

This is why the zkSync operation prices may differ significantly from similar ones on Ethereum.

#

Intrinsic Costs

Unlike Ethereum, where the intrinsic cost of transactions (21000 gas) is used to cover the price of updating the balances of the users, the nonce and signature verification, on zkSync these prices arenot included in the intrinsic costs for transactions.

This is due to the native support of account abstraction, where each account type may have their own transaction cost. Some may also use more zk-friendly signature schemes or other kinds of optimizations to lower transaction costs for their users.

zkSync transactions costs are used mostly to cover these different, intrinsic zero-knowledge proving costs. These can not be easily measured in code in real-time and are instead measured via testing and are hard coded.

Let's look at some of these hard coded constants:

#

Batch Overhead

In order to process the batch, the zkSync team has to pay the operational costs required by its proof. All of these costs we call “batch overhead”. It consists of two parts:

1. The L2 cost requirements for proving the circuits (denoted in L2 gas).
2. The L1 cost requirements for the proof verification as well as general batch processing (denoted in L1 gas).

The protocol aggregates as many transactions as possible into a single batch. Each transaction proportionally pays for the batch overhead in terms of how close the transaction brings the overall batch to beingsealed, i.e. closed and prepared for proof verification and submission on L1.

In the case of zkSync batches, here are the resources the protocol watches to decide when a batch is sealed:

1. Time.
2. The same as on Ethereum, the batch should generally not take too much time to be closed in order to provide better UX. To represent the time needed we use a batch gas limit, note that it is higher than the gas limit for a single transaction.
3. Slots for transactions.
4. The bootloader has a limited number of slots for transactions, i.e. it can not take more than a certain transactions per batch.
5. The memory of the bootloader.
6. The bootloader needs to store the transaction's ABI encoding in its memory & this fills it up. In practical terms, it serves as a penalty for having transactions with large calldata/signatures in case of custom accounts.
7. Pubdata bytes.
8. In order to fully appreciate the gains from the storage diffs, i.e. the fact that changes in a single slot happening in the same batch need to be published only once, we need to publish all the batch's public data only after the transaction has been processed. Right now, we publish all the data with the storage diffs as well as L2→L1 messages, etc in a single transaction at the end of the batch. Most nodes have limit of 128kb per transaction and so this is the limit that each zkSync batch should adhere to.

Each transaction spends the batch overhead proportionally to how much of these resources it requires.

Note that before the transaction is executed, the system can not know how many of these limited system resources the transaction will actually take. Therefore, we need to charge for the worst case and provide therefund at the end of the transaction.

#

baseFee

Like Ethereum, in order to protect us from DDoS attacks zkSync sets a limitedMAX_TRANSACTION_GAS_LIMIT per transaction.

Since the computation costs are relatively constant for us, we use abaseFee equal to the real costs for us to compute the proof for the corresponding 1 erg. Note thatgas_per_pubdata_limit should be then set high enough to cover the fees for the L1 gas needed to send a single pubdata byte on Ethereum. Under large L1 gas demands,gas_per_pubdata_limit would also need be large. That means thatMAX_TRANSACTION_GAS_LIMIT/gas_per_pubdata_limit could become too low to allow for enough pubdata for common use cases.

Therefore, to make common transactions always executable, we must enforce that the users are always able to send at leastGUARANTEED_PUBDATA_PER_TX bytes of pubdata in their transaction. Because of that, the neededgas_per_pubdata_limit for transactions should never grow beyondMAX_TRANSACTION_GAS_LIMIT/GUARANTEED_PUBDATA_PER_TX .

Setting a hard bound ongas_per_pubdata_limit also means that with the growth of L1 gas prices, the L2baseFee will have to grow as well. This ensures thatbase_fee * gas_per_pubdata_limit = L1_gas_price * l1_gas_per_pubdata .

This mainly only impacts for computationally intensive tasks. For these kinds of transactions, we need to conservatively charge a big upfront payment. However, it is compensated with a refund at the end of the transaction for all the overspent gas.

#

Trusted Gas Limit

While it was mentioned above that theMAX_TRANSACTION_GAS_LIMIT is needed to protect the operator from users stalling the state keeper by using too much computation, in case the users may need to use a lot of pubdata. For instance, to publish the bytecode of a new contract.

In this case, the required gasLimit may go beyond theMAX_TRANSACTION_GAS_LIMIT since the contracts can be tens of kilobytes in size.

Therefore, all the new published contracts are included as separate part of the pubdata: the factory dependencies field of

the transaction. This way the operator knows how much pubdata will have to published and how much gas they will have to spend on it.

That's why, to provide the better UX for users, the operator may provide the[rusted gas limitopen in new window](), i.e. the limit which exceedsMAX_TRANSACTION_GAS_LIMIT assuming that the operator is sure that the excess gas will be spent on the pubdata.

#

Refunds

Another distinctive feature of the fee model used on zkSync is the abundance of refunds. Refunds can be issued for:

- Unused limited system resources.
- Overpaid computation.

This is needed because of the relatively big upfront payments we discussed earlier required in zkSync to provide DDoS security.

#

Formulas and Constants for Calculating Fees

After determining price for each opcode in gas according to the model above, the following formulas are to be used for calculatingbaseFee andgasPerPubdata for a batch.

#

System-Wide Constants

These constants are to be hardcoded and can only be changed via either system contracts/bootloader or VM upgrade.

BATCH_OVERHEAD_L1_GAS (L_1_O)— The L1 gas overhead for a batch (proof verification, etc).

L1_GAS_PER_PUBDATA_BYTE (L_1_PUB) — The number of L1 gas needed for a single pubdata byte. It is slightly higher than 16 gas needed for publishing a non-zero byte of pubdata on-chain (currently the value of 17 is used).

BATCH_OVERHEAD_L2_GAS (EO)— The constant overhead denominated in gas. This overhead is created to cover the amortized costs of proving.

BLOCK_GAS_LIMIT (B) — The maximum number of computation gas per batch. This is the maximal number of gas that can be spent within a batch. This constant is rather arbitrary and is needed to prevent transactions from taking too much time from the state keeper. It can not be larger than the hard limit of 2^32 of gas for VM.

MAX_TRANSACTION_GAS_LIMIT (TM) — The maximal transaction gas limit. Fori -th single instance circuit, the price of each of its units isS C i = ⌈ T M C C i ⌉ SC_i = \lceil \frac{T_M}{CC_i} \rceil S C i

= ⌈ C C i

T M ⌉ to ensure that no transaction can run out of these single instance circuits.

MAX_TRANSACTIONS_IN_BATCH (TXM) — The maximum number of transactions per batch. A constant in bootloader. Can contain almost any arbitrary value depending on the capacity of batch that we want to have.

BOOTLOADER_MEMORY_FOR_TXS (BM) — The size of the bootloader memory that is used for transaction encoding (i.e. excluding the constant space, preallocated for other purposes).

GUARANTEED_PUBDATA_PER_TX (PG) — The guaranteed number of pubdata that should be possible to pay for in one zkSync batch. This is a number that should be enough for most reasonable cases.

#

Derived Constants

Some of the constants are derived from the system constants above:

MAX_GAS_PER_PUBDATA (EPMax) — thegas_price_per_pubdata that should always be enough to cover for publishing a pubdata byte:

E P M a x = ⌊ T M P G ⌋ EP_{Max} = \lfloor \frac{T_M}{P_G} \rfloor E P M a x

= ⌊ P G

T M ⌋

#

Externally-Provided Batch Parameters

L1_GAS_PRICE (L_1_P) — The price for L1 gas in ETH.

FAIR_GAS_PRICE (Ef) — The “fair” gas price in ETH, that is, the price of proving one circuit (in Ether) divided by the number we chose as one circuit price in gas.

E f = P r i c e C E C E_f = \frac{Price_C}{E_C} E f

= E C

P r i c e C

wherePriceC is the price for proving a circuit in ETH. Even though this price will generally be volatile (due to the volatility of ETH price), the operator is discouraged to change it often, because it would volatile both volatile gas price and (most importantly) the requiredgas_price_per_pubdata for transactions.

Both of the values above are currently provided by the operator. Later on, some decentralized/deterministic way to provide these prices will be utilized.

#

Determiningbase_fee

When the batch opens, we can calculate theFAIR_GAS_PER_PUBDATA_BYTE (EPf) — “fair” gas per pubdata byte:

E P f = ⌈ L 1 p ∗ L 1 P U B E f ⌉ EP_f = \lceil \frac{L1_p * L1_{PUB}}{E_f} \rceil E P f

= ⌈ E f

L 1 p

∗

L 1 P U B ⌉

There are now two situations that can be observed:

I.

E P f

E P M a x EP_f > EP_{Max} E P f

E P M a x

This means that the L1 gas price is so high that if we treated all the prices fairly, then the number of gas required to publish guaranteed pubdata is too high, i.e. allowing at leastPG pubdata bytes per transaction would mean that we would to supporttx .gasLimit greater that the maximum gas per transactionTM , allowing to run out of other finite resources.

IfE P f

E P M a x EP_f > EP_{Max} E P f

E P M a x , then the user needs to artificially increase the providedEf to bring the neededtx .gasPerPubdataByte toEPmax

In this case we set the EIP1559baseFee (Base):

B a s e = m a x (E f , ⌈ L 1 P ∗ L 1 P U B E P m a x ⌉) Base = max(E_f, \lceil \frac{L1_P * L1_{PUB}}{EP_{max}} \rceil) B a se

= ma x (E f ,

⌈ E P ma x

L 1 P

*

L 1 P U B])

Only transactions that have at least this high gasPrice will be allowed into the batch.

II.

Otherwise, we keepB a s e * = E f Base* = E_f B a se *

= E f

Note, that both cases are covered with the formula in case (1), i.e.:

B a s e = m a x (E f , ⌈ L 1 P * L 1 P U B E P m a x ⌉) Base = max(E_f, \lceil \frac{L1_P * L1_{PUB}}{EP_{max}} \rceil) B a se

= ma x (E f ,

⌈ E P ma x

L 1 P

*

L 1 P U B])

This is the base fee that will be always returned from the API viaeth_gasGasPrice .

#

Overhead for a Transaction

Let's define bytx .actualGasLimit as the actual gasLimit that is to be used for processing of the transaction (including the intrinsic costs). In this case, we will use the following formulas for calculating the upfront payment for the overhead:

S O = 1 / T X M S_O = 1/TX_M S O

= 1/ T X M

M O (t x) = e n c L e n (t x) / B M M_O(tx) = encLen(tx) / B_M M O (t x)

= e n c L e n (t x) / B M

E A O (t x) = t x . a c t u a l G a s L i m i t / T M E_{AO}(tx) = tx.actualGasLimit / T_M E A O (t x)

= t x . a c t u a lG a s L imi t / T M

O (t x) = m a x (S O , M O (t x) , E O (t x)) O(tx) = max(S_O, M_O(tx), E_O(tx)) O (t x)

= ma x (S O ,

M O (t x) ,

E O (t x))

where:

SO — is the overhead for taking up 1 slot for a transaction

MO (tx) — is the overhead for taking up the memory of the bootloader

encLen (tx) — the length of the ABI encoding of the transaction's struct.

EAO (tx) — is the overhead for potentially taking up the gas for single instance circuits.

O (tx) — is the total share of the overhead that the transaction should pay for.

Then we can calculate the overhead that the transaction should pay as the following one:

L 1 O (t x) = ⌈ L 1 O L 1 P U B ⌉ * O (t x)

E O (t x) = E O * O (t x) L1_O(tx) = \lceil \frac{L1_O}{L1_{PUB}} \rceil * O(tx) \ E_O(tx) = E_O * O(tx) L 1 O (t x)

= ⌈ L 1 P U B

L 1 O ⌉

∗ O (t x)

E O (t x)

= E O

∗ O (t x)

Where

L_1_O (tx) — the number of L1 gas overhead (in pubdata equivalent) the transaction should compensate for gas.

EO (tx) — the number of L2 gas overhead the transaction should compensate for.

Then:

overhead _gas (tx) =EO (tx) +tx .gasPerPubdata ·L_1_O (tx)

When a transaction is being estimated, the server returns the following gasLimit:

tx .gasLimit =tx .actualGasLimit +overhead _gas (tx)

Note, that when the operator receives the transaction, it knows onlytx .gasLimit . The operator could derive theoverhead___gas (tx) and provide the bootloader with it. The bootloader will then derivetx .actualGasLimit =tx .gasLimit –overhead __gas*(_tx*) and use the formulas above to derive the overhead that the user should've paid under the derivedtx .actualGasLimit to ensure that the operator does not overcharge the user.

#

overhead (tx)

For the ease of integer calculation, we will use the following formulas to derive theoverhead (tx):

B O (t x) = E O + t x . g a s P e r P u b d a t a B y t e · ⌊ L 1 O L 1 P U B ⌋ B_O(tx) = E_O + tx.gasPerPubdataByte \cdot \lfloor \frac{L1_O}{L1_{PUB}} \rfloor B O (t x)

= E O

+ t x . g a s P er P u b d a t a B y t e

· ⌊ L 1 P U B

L 1 O ⌋

B O B_O B O denotes the overhead for batch in gas that the transaction would have to pay if it consumed the resources for entire batch.

Then,overhead _gas (tx) is the maximum of the following expressions:

1. S
2. O
3. =
4. ⌈
5. B
6. O
7. T
8. X
9. M
10. ⌉
11. S_O = \lceil \frac{B_O}{TX_M} \rceil
12. S
13. O
14.
15. =
16. ⌈
17. T
18. X
19. M

20.
21. B
22. O
23.
24.
25.]
26. M
27. O
28. (
29. t
30. x
31.)
32. =
33. ⌈
34. B
35. O
36. ·
37. e
38. n
39. c
40. o
41. d
42. i
43. n
44. g
45. L
46. e
47. n
48. (
49. t
50. x
51.)
52. B
53. M
54. ⌉
55. M_O(tx) = \lceil \frac{B_O \cdot encodingLen(tx)}{B_M} \rceil
56. M
57. O
58.
59. (
60. t
61. x
62.)
63. =
64. ⌈
65. B
66. M
67.
68. B
69. O
70.
71. ·
72. e
73. n
74. co
75. d
76. in
77. gL
78. e
79. n
80. (
81. t
82. x
83.)
84.
85. ⌉
86. E
87. O

(
t
x
)
=
⌈
B
O
·
t
x
.
g
a
s
B
o
d
y
L
i
m
i
t
T
M
⌉
E_O(tx) = \lceil \frac{B_O \cdot tx.gasBodyLimit}{T_M} \rceil
E
O
(
t
x
)
=
⌈
T
M
B
O
·
t
x
.
g
a
s
B
o
d
y
L
imi
t
⌉

#

Derivingoverhead_gas(tx) fromtx.gasLimit

The task that the operator needs to do is the following:

Given the tx.gasLimit, it should find the maximaloverhead_gas(tx) , such that the bootloader will accept such transaction, that is, if we denote byOop the overhead proposed by the operator, the following equation should hold:

O o p ≤ o v e r h e a d g a s (t x) O_{op} ≤ overhead_gas(tx) O o p

≤ o v er h e a d g a s (t x)

for thet x . b o d y G a s L i m i t tx.bodyGasLimit t x . b o d y G a s L imi t we use thet x . b o d y G a s L i m i t tx.bodyGasLimit t x . b o d y G a s L imi t =t x . g a s L i m i t – O o p tx.gasLimit – O_{op} t x . g a s L imi t

– O o p .

There are a few approaches that could be taken here:

- Binary search. However, we need to be able to use this formula for the L1 transactions too, which would mean that binary search is too costly.
- The analytical way. This is the way that we will use and it will allow us to find such an overhead in O(1), which is acceptable for L1->L2 transactions.

Let's rewrite the formula above the following way:

O o p ≤ m a x (S O , M O (t x) , E O (t x)) O_{op} ≤ max(SO, MO(tx), EO(tx)) O o p

≤ ma x (SO ,

MO (t x) ,

EO (t x))

So we need to find the maximalO o p O_{op} O o p , such thatO o p ≤ m a x (S O , M O (t x) , E O (t x)) O_{op} ≤ max(S_O, M_O(tx), E_O(tx)) O o p

≤ ma x (S O ,

M O (t x) ,

E O (t x)) . Note, that it means ensuring that at least one of the following is true:

1. O
2. o
3. p
4. ≤
5. S
6. O
7. O_{op} ≤ S_O
8. O
9. o
10. p
11.
12. ≤
13. S
14. O
15.
16. O
17. o
18. p
19. ≤
20. M
21. O
22. (
23. t
24. x
25.)
26. O_{op} ≤ M_O(tx)
27. O
28. o
29. p
30.
31. ≤
32. M
33. O
34.
35. (
36. t

37. x
38.)
39. O
40. o
41. p
42. ≤
43. E
44. O
45. (
46. t
47. x
48.)
49. O_{op} ≤ E_O(tx)
50. O
51. o
52. p
53.
54. ≤
55. E
56. O
57.
58. (
59. t
60. x
61.)

So let's find the largestOop for each of these and select the maximum one.

- Solving for (1)

O o p = ⌈ B O T X M ⌉ O_{op} = \lceil \frac{B_O}{TX_M} \rceil O o p

= ⌈ T X M

B O ⌉

- Solving for (2)

O o p = ⌈ e n c L e n (t x) ⋅ B O B M ⌉ O_{op} = \lceil \frac{encLen(tx) \cdot B_O}{B_M} \rceil O o p

= ⌈ B M

e n c L e n (t x)

.

B O ⌉

- Solving for (3)

This one is somewhat harder than the previous ones. We need to find the largestO_{op} , such that:

O o p ≤ ⌈ t x . a c t u a l E r g s L i m i t ⋅ B O T M ⌉ O_{op} \le \lceil \frac{tx.actualErgsLimit \cdot B_O}{T_M} \rceil \ O o p

≤ ⌈ T M

t x . a c t u a lE r g s L imi t

.

B O ⌉

O o p ≤ ⌈ (t x . e r g s L i m i t – O o p) ⋅ B O T M ⌉ O_{op} \le \lceil \frac{(tx.ergsLimit - O_{op}) \cdot B_O}{T_M} \rceil \ O o p

≤ ⌈ T M

(t x . er g s L imi t

–

O o p)

.

B O ⌉

O o p ≤ ⌈ B O · (t x . e r g s L i m i t – O o p) T M ⌉ O_{op} ≤ \lceil \frac{B_O \cdot (tx.ergsLimit - O_{op})}{T_M} \rceil O o p

≤ ⌈ T M

B O

.

(t x . er g s L imi t

–

O o p) ⌉

Note, that all numbers here are integers, so we can use the following substitution:

O o p – 1 < (t x . e r g s L i m i t – O o p) · B O T M O_{op} -1 \lt \frac{(tx.ergsLimit - O_{op}) \cdot B_O}{T_M} \ O o p

– 1

< T M

(t x . er g s L imi t

–

O o p)

.

B O

(O o p – 1) T M < (t x . e r g s L i m i t – O o p) · B O (O_{op} -1)T_M \lt (tx.ergsLimit - O_{op}) \cdot B_O \ (O o p

– 1) T M

< (t x . er g s L imi t

– O o p)

· B O

O o p T M + O o p B O < t x . e r g s L i m i t · B O + T M O_{op} T_M + O_{op} B_O \lt tx.ergsLimit \cdot B_O + T_M \ O o p
T M

+ O o p B O

< t x . er g s L imi t

· B O

+ T M

O o p < t x . e r g s L i m i t · B O + T M B O + T M O_{op} \lt \frac{tx.ergsLimit \cdot B_O + T_M}{B_O + T_M} \ O o p

< B O

+

T M

t x . er g s L imi t

.

B O

+

T M

Meaning, in other words:

O o p = ⌊ t x . e r g s L i m i t · B O + T M − 1 B O + T M ⌋ O_{op} = \lfloor \frac{tx.ergsLimit \cdot B_O + T_M - 1}{B_O + T_M} \rfloor O o p

= ⌊ B O

+

T M

t x . er g s L imi t

.

B O

+

T M

−

1 ⌋

Then, the operator can safely choose the largest one.

#

Discounts by the operator

It is important to note that the formulas provided above are to withstand the worst-case scenario and these are the formulas used for L1->L2 transactions (since these are forced to be processed by the operator). However, in reality, the operator typically would want to reduce the overhead for users whenever it is possible. For instance, in the server, we underestimate the maximal potentialMAX_GAS_PER_TRANSACTION , since usually the batches are closed because of either the pubdata limit or the transactions' slots limit. For this reason, the operator also provides the operator's proposed overhead. The only thing that the bootloader checks is that this overhead isnot larger than the maximal required one. But the operator is allowed to provide a lower overhead.

#

Refunds

As you could see above, this fee model introduces quite some places where users may overpay for transactions:

- For the pubdata when L1 gas price is too low
- For the computation when L1 gas price is too high
- The overhead, since the transaction may not use the entire batch resources they could.

To compensate users for this, we will provide refunds for users. For all of the refunds to be provable, the counter counts the number of gas that was spent on pubdata (or the number of pubdata bytes published). We will denote this number bypubdataused . For now, this value can be provided by the operator.

The fair price for a transaction is

F a i r F e e = E f · t x . c o m p u t a t i o n a l G a s + E P f · p u b d a t a u s e d FairFee = E_f \cdot tx.computationalGas + EP_f \cdot pubdataused F ai r F ee

= E f

· t x . co m p u t a t i o na lG a s

+ E P f

· p u b d a t a u se d

We can derivet x . c o m p u t a t i o n a l G a s = g a s s p e n t − p u b d a t a u s e d · t x . g a s P r i c e P e r P u b d a t a tx.computationalGas = gasspent − pubdataused \cdot tx.gasPricePerPubdata t x . co m p u t a t i o na lG a s

= g a ss p e n t

− p u b d a t a u se d

· t x . g a s P r i ce P er P u b d a t a , wheregasspent is the number of gas spent for the transaction (can be trivially fetched in Solidity).

Also, theFairFee will include the actual overhead for batch that the users should pay for.

The fee that the user has actually spent is:

A c t u a l F e e = g a s s p e n t · g a s P r i c e ActualFee = gasspent \cdot gasPrice A c t u a lF ee

= g a ss p e n t

· g a s P r i ce

So we can derive the overpay as

A c t u a l F e e – F a i r F e e ActualFee – FairFee A c t u a lF ee

– F ai r F ee

In order to keep the invariant ofg a s U s e d · g a s P r i c e = f e e gasUsed \cdot gasPrice = fee g a s U se d

· g a s P r i ce

= f ee , we will formally refundA c t u a l F e e – F a i r F e e B a s e \frac{ActualFee - FairFee}{Base} B a se

A c t u a lF ee – F ai r F ee gas.

At the moment, this counter is not accessible within the VM and so the operator is free to provide any refund it wishes (as long as it is greater than or equal to the actual amount of gasLeft after the transaction execution).

#

Refunds for repeated writes

zkEVM is a statediff-based rollup, i.e. the pubdata is published not for transactions, but for storage changes. This means that whenever a user writes into a storage slot, he incurs certain amount of pubdata. However, not all writes are equal:

- If a slot has been already written to in one of the previous batches, the slot has received a short id, which allows it to require less pubdata in the state diff.
- Depending on thevalue
- written into a slot, various compression optimizations could be used and so we should reflect that too.
- Maybe the slot has been already written to in this batch and so we don't to charge anything for it.

You can read more about how we treat the pubdata hereopen in new window .

The important part here is that while such refunds are inlined (i.e. unlike the refunds for overhead they happen in-place during execution and not after the whole transaction has been processed), they are enforced by the operator. Right now, the operator is the one that decides what refund to provide.

#

Improvements in the upcoming releases

The fee model explained above, while fully functional, has some known issues. These will be tackled with the following upgrades.

#

The quadratic overhead for pubdata

Note, that the computational overhead is proportional to thetx.gasLimit and the amount of funds the user will have to pay is proportional to the L1 gas price (recall the formula ofB_O). We can roughly express the transaction overhead from computation astx.gasLimit * L1_GAS_PRICE * C whereC is just some constant. Note, that since a transaction typically contains some storage writes, and itstx.gasLimit = gasSpentOnCompute + pubdataPublished * gasPricePerPubdata ,tx.gasLimit is roughly proportional togasPricePerPubdata and so it is also proportional toL1_GAS_PRICE .

This means that formulatx.gasLimit * L1_GAS_PRICE * C becomesquadratic to theL1_GAS_PRICE .

#

gasUsed depends togasLimit

While in general it shouldn't be the case assuming the correct implementation of refunds , in practice it turned out that the formulas above, while robust, estimate for the worst case which can be very difference from the average one. In order to improve the UX and reduce the overhead, the operator charges less for the execution overhead. However, as a compensation for the risk, it does not fully refund for it.

#

L1->L2 transactions do not pay for their execution on L1

TheexecuteBatches operation on L1 is executed inO(N) where N is the number of priority ops that we have in the batch. Each executed priority operation will be popped and so it incurs cost for storage modifications. As of now, we do not charge for it.

#

zkEVM Fee Components (Revenue & Costs)

- On-Chain L1 Costs *L1 Commit Batches* The commit batches transaction submits pubdata (which is the list of updated storage slots) to L1. The cost of a commit transaction is calculated asconstant overhead + price of pubdata
-
    -
        - . Theconstant overhead
-
    -
        - cost is evenly distributed among L2 transactions in the L1 commit transaction, but only at higher transaction loads. As for theprice of pubdata
-
    -
        - , it is known how much pubdata each L2 transaction consumed, therefore, they are charged directly for that. Multiple L1 batches can be included in a single commit transaction.
-
    - L1 Prove Batches* Once the off-chain proof is generated, it is submitted to L1 to make the rollup batch final. Currently, each proof contains only one L1 batch.
-
    - L1 Execute Batches* The execute batches transaction processes L2 -> L1 messages and marks executed priority operations as such. Multiple L1 batches can be included in a single execute transaction.
-
    - L1 Finalize Withdrawals* While not strictly part of the L1 fees, the cost to finalize L2 → L1 withdrawals are covered by Matter Labs. The finalize withdrawals transaction processes user token withdrawals from zkEVM to Ethereum. Multiple L2 withdrawal transactions are included in each finalize withdrawal transaction.
- On-Chain L2 Revenue *L2 Transaction Fee* This fee is what the user pays to complete a transaction on zkEVM. It is calculated asgasLimit x baseFeePerGas - refundedGas x baseFeePerGas
-
    -
        - , or more simply,gasUsed x baseFeePerGas
-
    -
        - .
- Profit = L2 Revenue - L1 Costs - Off Chain Infrastructure Costs

Prev Blocks Next Finality