MEV-resistant ZK-Rollups with Practical VDE (PVDE)

@zeroknight , @0xTariz , and @radzin from Radius.xyz

Abstract

Current MEV solutions based on time-lock puzzles are not viable. Operators cannot easily detect invalid time-lock puzzles or transactions, which can even lead to DoS attacks. We are designing an MEV-resistant ZK-Rollup with a practical

approach to MEV minimization using Practical Verifiable Delay Encryption (PVDE). This method generates zk proofs within 5 seconds, a validity proof needed to prove that solving the time-lock puzzles will lead to the correct decryption of valid transactions.

Background & Motivation

Decentralization Is Disguised

The structure

of a blockchain is fully decentralized, but the content

of each block is not.

MEV attacks occur from revealed transaction data and the centralization

of miners who see the contents

of block transactions and decide which transactions are included in a block. By leveraging their power to censor and reorder block transactions, miners can perform front-running, back-running, or sandwiching attacks to extract profit from users. The problem is that most users are unaware of being the victim of these malicious attacks.

In L1, block-building competitions among miners of different blockchains and high gas fees can prevent MEV exploitations from miners/attackers. Proposer/Builder Separation (PBS) is another proposed solution for MEVs in L1s, but users' assets are still prone to attacks as MEV opportunities are still publicized in a decentralized way.

In L2s, the case is a bit different. If the operator correctly computes the state transitions and includes only valid transactions, it becomes difficult to detect the censorships or intentional reorderings as only the computational integrity is verified. This creates a debate to the centralization/decentralization dichotomy. Due to the limited nature of blockchain scalability, L2 needs to maintain a small number of operators for its scalability which allows more room for operators to extract MEV profit. Additionally, lower gas fees on L2s attract mempool searchers (attackers) to MEV attacks.

We must assume that operators work in a permissionless

manner to ensure a transparent and safe L2 ecosystem. A grounded approach that prevents the manipulation of blocks by the centralized operators in L2 is therefore needed. If there is no inherent solution to control the significant power of operators,

The adoption of layer 2 scaling solutions could represent the centralization of control over assets stored within them. — barryWhiteHat

MEV auctions were proposed as a scheme to decentralize the operators. But asbarryWhiteHat mentions, we need to be more cautious in the way we elect the operators through consensus mechanisms or with economical schemes like PBS as the security of L2s might very well be served. To prevent the users' assets from being exposed to MEV profits, we propose a complete

privacy method for MEV minimization.

Time-lock Puzzle

Our MEV-resistant L2 solution ensures complete

privacy of transactions as it reveals the contents of the transaction only after the transaction order is determined by the operator. We achieve this by encrypting the transactions temporarily based on time-lock puzzle. This scheme delays the time for the operator to find the symmetric key used to decrypt the transaction.

[Create and Encrypt Time-lock Puzzle]

The trader generates a time-lock puzzle that finds the symmetric key, then encrypts the transaction using the symmetric key.

1. Create time-lock puzzle
2. Generate a modulus: N = p*q
3. Select generator: g\in G, \ \ where\ \ G \ \ is \ \ RSA \ \ group
4. Compute symmetric key: S_K = g^{2^T} \ \ mod \ \ N
5. Generate a modulus: N = p*q
6. Select generator: g\in G, \ \ where\ \ G \ \ is \ \ RSA \ \ group
7. Compute symmetric key: S_K = g^{2^T} \ \ mod \ \ N
8. Encrypt transaction
9. Generate a symmetric key on an elliptic curve

k = (k_0, k_1) \leftarrow JubJubAffine(S_K) \in F^2_q

1. Encrypt message with Poseidon encryption scheme: C_{TX} = ENC(k, TX)
2. Generate a symmetric key on an elliptic curve

k = (k_0, k_1) \leftarrow JubJubAffine(S_K) \in F^2_q

1. Encrypt message with Poseidon encryption scheme: C_{TX} = ENC(k, TX)

[Solve and Decrypt Time-lock Puzzle]

The operator receives the time-lock puzzle from the trader and solves it to find the symmetric key, then decrypts the transaction using the symmetric key.

1. Solve time-lock puzzle
2. Receive public parameter: N, T
3. Compute symmetric key: S_K = g^{2^T} \ \ mod \ \ N
4. Receive public parameter: N, T
5. Compute symmetric key: S_K = g^{2^T} \ \ mod \ \ N
6. Decrypt cipher text
7. Generate a symmetric key on an elliptic curve

k = (k_0, k_1) \leftarrow JubJubAffine(S_K) \in F^2_q

1. Decrypt message with Poseidon encryption scheme: TX = DEC(C_{TX}, k)
2. Generate a symmetric key on an elliptic curve

k = (k_0, k_1) \leftarrow JubJubAffine(S_K) \in F^2_q

1. Decrypt message with Poseidon encryption scheme: TX = DEC(C_{TX}, k)

Practical Verifiable Delay Encryption (PVDE)

Time lock puzzle schemes require large computational resources from operators. If a trader sends an invalid puzzle to the operator, resources are largely and meaninglessly wasted which can also lead to DoS attacks by malicious traders.

To prevent this, the trader must generate a zk proof to prove the integrity of the time-lock puzzle before the operator solves the puzzle.

The trader’s statement is as follows:

\pi_{PVDE}

: The output value (also the symmetric key used for the decryption of the encrypted transaction) is found by computing the time-lock puzzle 2^ttimes

The circuit must include the two computations to prove the statement:

1. Time-lock puzzle

: g^{2^{T}} mod \ \ N = S_K

1. Poseidon Encryption

: ENC(TX, S_K) → C_{TX}

It is difficult to solve the current RSA-based naive time-lock puzzle inside the zk circuit. For the past few months, we have been working to find a solution to the RSA sequential computation inside the zk circuit, with the help of Ethereum Foundation. To do this, we designed a total of 4 circuit computations and measured the proof generation time.

Our Practical Verifiable Delay Encryption (PVDE) method successfully generates the zk proof for the RSA-based time-lock puzzle within 5 seconds. This is an applicable and practical approach to generating the proof, which proves that solving the time-lock puzzle will lead to the correct decryption of valid transactions.

See the results of the experiment below. A more detailed experiment methods and results with the algorithm will be revealed on ethresear.ch.

Many thanks to @barryWhiteHat and @Wanseob-Lim from Ethereum Foundation

Spec

- CPU: 2-way E5-2683 v4 (2.1GHz, 16-core)
- RAM: 64 GB
- Storage: 512 GB (SSD)

[

1

898×106 10.3 KB

](https://ethresear.ch/uploads/default/original/2X/5/52b17e0fb6c0a9e4ae87b51ce9fece9ee65de7e0.png)

Circuit design (RSA: 2048 bits, T: 32678)

(When T is set to 32678, the operator solves the time-lock puzzle g^{2^T}

in 7 to 8 seconds.)

1. Naive time-lock puzzle
2. Proving time: N/A
3. Scheme
4. g^{2^{T}} mod \ \ N = S_K
5. g^{2^{T}} mod \ \ N = S_K
6. Proving time: N/A
7. Scheme
8. g^{2^{T}} mod \ \ N = S_K
9. g^{2^{T}} mod \ \ N = S_K
10. Trapdoor time-lock puzzle
11. Proving time: N/A
12. Scheme
13. Select random prime number p,q
14. Compute trapdoor \phi(N) = (p-1)(q-1)
15. 2^T \ \ mod \ \ \phi(N) = r

16. g^r \mod \ \ N = S_K
17. Select random prime number p,q
18. Compute trapdoor \phi(N) = (p-1)(q-1)
19. 2^T \ \ mod \ \ \phi(N) = r
20. g^r \mod \ \ N = S_K
21. Proving time: N/A
22. Scheme
23. Select random prime number p,q
24. Compute trapdoor \phi(N) = (p-1)(q-1)
25. 2^T \ \ mod \ \ \phi(N) = r
26. g^r \mod \ \ N = S_K
27. Select random prime number p,q
28. Compute trapdoor \phi(N) = (p-1)(q-1)
29. 2^T \ \ mod \ \ \phi(N) = r
30. g^r \mod \ \ N = S_K
31. Efficient time-lock puzzle (our new scheme)
32. Proving time: 540 secs
33. Scheme
34. generate parameters via a trusted setup : (g, g^{2^T})
35. choose a random secret s

(128 bits)

1. generate inputs for zk-SNARKs (s_1, s_2) \leftarrow (g^s,(g^{2^T})^s)
2. s_2

is used as a symmetric key and can be obtained via (s_1)^{2^T}

1. s_2

is used as a symmetric key and can be obtained via (s_1)^{2^T}

1. generate a validity proof \pi \leftarrow \Pi.prove(g, g^{2^T}, s_1; s, s_2)
2. verify the proof r \leftarrow \Pi.verify(\pi, g, g^{2^T}, s_1)
3. obtain the symmetric key via time lock puzzle : (s_1)^{2^T} = S_K
4. generate parameters via a trusted setup : (g, g^{2^T})
5. choose a random secret s

(128 bits)

1. generate inputs for zk-SNARKs (s_1, s_2) \leftarrow (g^s,(g^{2^T})^s)
2. s_2

is used as a symmetric key and can be obtained via (s_1)^{2^T}

1. s_2

is used as a symmetric key and can be obtained via (s_1)^{2^T}

1. generate a validity proof \pi \leftarrow \Pi.prove(g, g^{2^T}, s_1; s, s_2)
2. verify the proof r \leftarrow \Pi.verify(\pi, g, g^{2^T}, s_1)
3. obtain the symmetric key via time lock puzzle : (s_1)^{2^T} = S_K
4. Proving time: 540 secs
5. Scheme
6. generate parameters via a trusted setup : (g, g^{2^T})
7. choose a random secret s

(128 bits)

1. generate inputs for zk-SNARKs (s_1, s_2) \leftarrow (g^s,(g^{2^T})^s)
2. s_2

is used as a symmetric key and can be obtained via (s_1)^{2^T}

1. s_2

is used as a symmetric key and can be obtained via (s_1)^{2^T}

1. generate a validity proof \pi \leftarrow \Pi.prove(g, g^{2^T}, s_1; s, s_2)
2. verify the proof r \leftarrow \Pi.verify(\pi, g, g^{2^T}, s_1)
3. obtain the symmetric key via time lock puzzle : (s_1)^{2^T} = S_K
4. generate parameters via a trusted setup : (g, g^{2^T})
5. choose a random secret s

(128 bits)

1. generate inputs for zk-SNARKs (s_1, s_2) \leftarrow (g^s,(g^{2^T})^s)
2. s_2

is used as a symmetric key and can be obtained via (s_1)^{2^T}

1. s_2

is used as a symmetric key and can be obtained via (s_1)^{2^T}

1. generate a validity proof \pi \leftarrow \Pi.prove(g, g^{2^T}, s_1; s, s_2)
2. verify the proof r \leftarrow \Pi.verify(\pi, g, g^{2^T}, s_1)
3. obtain the symmetric key via time lock puzzle : (s_1)^{2^T} = S_K
4. PVDE (our new scheme)
5. Proving time: 5 secs
6. Scheme: Combined zk-SNARK and sigma protocol
7. Proving time: 5 secs
8. Scheme: Combined zk-SNARK and sigma protocol

Participants

We design a ZK-Rollup solution to MEV exploitations caused by the centralization of operators producing the contents of the block and prevent the traders' transactions and assets from being exposed to the attackers.

Trader

Traders are parties that generate and execute L2 transactions.

To prevent assets from being exposed to the operators, the trader encrypts the transaction, then sends the time-lock puzzle, encrypted transaction, and zk proof to the operator.

1. Generate a transaction
2. Encrypt the transaction with the symmetric key generated with the time-lock puzzle
3. Generate a zk-SNARK proof to verify the integrity of the time-lock puzzle and the encrypted transaction
4. Send the time-lock puzzle, encrypted transaction, and zk proof to the operator

Operator

Operators are parties that collect and execute L2 transactions.

Operators can censor and intentionally reorder the traders' transactions to extract additional profit for themselves. To prevent these malicious MEV activities, operators must determine the block transaction order and send the corresponding commitment to the trader before they can decrypt the transactions and see the contents.

1. Verify that the trader's time-lock puzzle is valid
2. Start solving the time-lock puzzle to find the symmetric key (sequentially compute 2^T

)

1. At the same time, determine the block transaction order and send the corresponding commitment to the trader. This should be completed before solving the time-lock puzzle
2. Decrypt the transaction using the symmetric key and execute the transactions in the commitment

Architecture

[

2

960×540 41.9 KB

](https://ethresear.ch/uploads/default/original/2X/3/37e6fdaa6fdfeaf9c05433374745a19d67b36950.jpeg)

Phase 1. Trader Generates Puzzle

1. Generate a transaction
2. Generate symmetric key with time-lock puzzle, encrypt the transaction, then generate validity proofs for the puzzle
3. Setup(Λ

) → pp = (g, N, T)

1. Generate_puzzle(pp, TX

) → (ENC_{tx}, \pi_{PVDE})

1. Time-lock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Encryption(S_K, TX

) → C_{TX}

C_{TX} = ENC(S_K, TX)

1. Generate_proof(pp,TX

) → \pi_{PVDE}

1. Time-lock_puzzle(pp

) → S_K

$S_K = g^{2^T} \ \ mod \ \ N$

1. Encryption(S_K, TX

) → C_{TX}

$C_{TX} = ENC(S_K, TX)$

1. Generate_proof(pp,TX

) → π_{PVDE}

1. Setup(Λ

) → pp = (g, N, T)

1. Generate_puzzle(pp, TX

) → (ENC_{tx}, π_{PVDE})

1. Time-lock_puzzle(pp

) → S_K

$S_K = g^{2^T} \ \ mod \ \ N$

1. Encryption(S_K, TX

) → C_{TX}

$C_{TX} = ENC(S_K, TX)$

1. Generate_proof(pp,TX

) → π_{PVDE}

1. Time-lock_puzzle(pp

) → S_K

$S_K = g^{2^T} \ \ mod \ \ N$

1. Encryption(S_K, TX

) → C_{TX}

$C_{TX} = ENC(S_K, TX)$

1. Generate_proof(pp,TX

) → π_{PVDE}

1. Trader sends the following to the operator:
2. Public parameters of time-lock puzzle
3. Encrypted transaction: C_{TX}
4. Transaction hash value: Hash(TX)
5. π_{PVDE}
6. Public parameters of time-lock puzzle
7. Encrypted transaction: C_{TX}
8. Transaction hash value: Hash(TX)
9. π_{PVDE}

Phase 2. Operator Determines Transaction Order

1. Trader verifies the validity of time-lock puzzle. If true, go to step 2.

2. Verify(\pi_{PVDE}

) → {True, False}

1. Verify(\pi_{PVDE}

) → {True, False}

1. Operator starts computing Timelock_puzzle to find the symmetric key for the decryption of the transaction
2. Timelock_puzzle(pp

)

1. Timelock_puzzle(pp

)

1. Determine the order of the transaction using Merkle Mountain Range(MMR) and send the commitment with order

to the trader 1. MMR(Hash(TX)) \rightarrow Commit(order, Hash(TX), merkle _\ root, merkle_\ path)

1. MMR(Hash(TX)) \rightarrow Commit(order, Hash(TX), merkle _\ root, merkle_\ path)

Phase 3. Trader Confirms Order

1. Trader receives the commitments and checks the order of the transaction, then confirms that the time to receive the commitment was less than the minimum time-lock puzzle computation time
2. Confirm that the transaction is included in Commit(order, Hash(TX), merkle _\ root, merkle_\ path)
3. Check: T_{response} - T_{order} < Δ
4. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle
2. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle
2. Confirm that the transaction is included in Commit(order, Hash(TX), merkle _\ root, merkle_\ path)
3. Check: T_{response} - T_{order} < Δ
4. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle
2. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle

Phase 4. Operator Decrypts Transaction and Executes

1. Operator solves the symmetric key with time-lock puzzle and decrypts the transaction
2. Solve_puzzle(pp, C_{TX}

) → TX

1. Timelock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Decryption(N, ENC_{tx}, S_K

) → TX

TX = DEX(C_{TX}, S_K)

1. Timelock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Decryption(N, ENC_{tx}, S_K

) → TX

TX = DEX(C_{TX}, S_K)

1. Transactions are executed in the order determined by MMR

2. Solve_puzzle(pp, C_{TX}

) → TX

1. Timelock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Decryption(N, ENC_{tx}, S_K

) → TX

TX = DEX(C_{TX}, S_K)

1. Timelock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Decryption(N, ENC_{tx}, S_K

) → TX

TX = DEX(C_{TX}, S_K)

1. Transactions are executed in the order determined by MMR

Open For Discussion

The problem of MEV is the centralization

of operators: the trader’s transactions and assets are pre-exposed to the operators in L2s. And as stated above, it is not possible to detect the censorships and intentional reorderings of transactions in L1s.

Our architecture is designed so that operators produce the block contents with encrypted transactions to prevent the traders’

transactions and assets from being pre-exposed. The operators then need to send the finalized transaction commitments to the traders. Traders can challenge operators that did not execute the finalized order of transactions.

Our team is also discussing ideas to verify the true fairness of operators on L1 - we'd be happy to discuss this with the ethresear.ch community!

Order Commitment Validity

[

3

960×540 46.3 KB

](https://ethresear.ch/uploads/default/original/2X/8/84506bb32cbcfe2e12af748dada325f44c26ee47.jpeg)

Order commitment validity ensures the L1 verifier contract validates that the operator executed the transactions in the determined order and finalizes the state without any challenges from the trader.

1. Trader checks the order of the transactions in the commitments and confirms that the time to send the commitment was less than the minimum time-lock puzzle computation time
2. Use membership proof to check that the trader's transaction is included in Commit(order, Hash(TX), merkle _\ root, merkle_\ path)
3. Check: T_{response} - T_{order} < Δ
4. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle
2. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle
2. Use membership proof to check that the trader's transaction is included in Commit(order, Hash(TX), merkle _\ root, merkle_\ path)
3. Check: T_{response} - T_{order} < Δ
4. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle
2. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle
2. T_{response}

: Total time for trader to send the encrypted tx and receive the commitment from the operator

1. T_{order}

: Expected time for the operator to determine the order and send the commitment to the trader

1. Δ = Total time to solve the time-lock puzzle
2. Send signature to operator: Sig(Commit(order, Hash(TX), merkle _\ root, merkle_\ path)

3. Operator collects trader’s signatures and generates confirmation stat

e in the order response time

1. Operator solves the time-lock puzzle and decrypts the transaction, then collects the transactions by the order and generates block proposal state
2. Solve_puzzle(pp, C_{TX}

) → TX

1. Timelock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Decryption(N, ENC_{tx}, S_K

) → TX

TX = DEX(C_{TX}, S_K)

1. Timelock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Decryption(N, ENC_{tx}, S_K

) → TX

TX = DEX(C_{TX}, S_K)

1. Block proposal state: TX, Commit(order, Hash(TX), merkle _\ root, merkle_\ path)

2. Solve_puzzle(pp, C_{TX}

) → TX

1. Timelock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Decryption(N, ENC_{tx}, S_K

) → TX

TX = DEX(C_{TX}, S_K)

1. Timelock_puzzle(pp

) → S_K

S_K = g^{2^T} \ \ mod \ \ N

1. Decryption(N, ENC_{tx}, S_K

) → TX

TX = DEX(C_{TX}, S_K)

1. Block proposal state: TX, Commit(order, Hash(TX), merkle _\ root, merkle_\ path)

2. Operator executes the decrypted transaction and generates zk proof to validate computational integrity on L1. Check the following two conditions in the circuit when generating zk proof:

3. Confirm transaction validity

4. Confirm that both order

and Hash(TX

) in both confirmation state and block proposal state are same

1. Confirm transaction validity
2. Confirm that both order

and Hash(TX

) in both confirmation state and block proposal state are same

1. When the state transition is complete, operator sends the new state and the validity proof to L1 verifier contract
2. L1 verifier contract validates the computational integrity and order integrity using the validity proof and finalizes the state

Conclusion

MEV results in huge financial losses for traders. Thus, a solution to MEV minimization is imperative at current state. Our PVDE scheme is a practical approach to encrypt transactions using time-lock puzzles, so that operators do not waste any computational resources that lead to DoS attacks. The MEV-resistant ZK-rollup solution will fully prevent MEV attacks using cryptographic methods and eliminate the centralization of operators to provide a fair and efficient trading environment.