Dice Protocol: Commit-Reveal Randomness Oracle for Robinhood Chain

Version 1.0 — July 2026

Abstract

Dice Protocol is verifiable commit-reveal randomness infrastructure on Robinhood Chain (chain ID 4663). A designated provider pre-commits to a hash chain and reveals values on demand. The contract combines the requester contribution with the provider reveal using Keccak256 and delivers the result through an optional callback. The current v10 deployment uses one provider and an exact fee of 0.000025 ETH per request.

---

1. Introduction

1.1 Problem Statement

Onchain applications — games, lotteries, NFT mints, fair randomized distribution mechanisms — require a source of randomness that is:

1. **Two-party**: Neither requester nor provider can unilaterally choose the output of a completed reveal.
2. **Verifiable**: The randomness can be independently verified after the fact.
3. **Available**: The randomness is delivered reliably and within a predictable timeframe.
4. **Economically viable**: The cost per request is low enough for widespread adoption.

Robinhood Chain applications need dedicated randomness infrastructure because deterministic contract execution and chain-controlled values are not suitable private randomness sources. Dice Protocol provides a live commit-reveal option for this use case.

1.2 Solution

Dice Protocol implements a commit-reveal scheme based on hash chains:

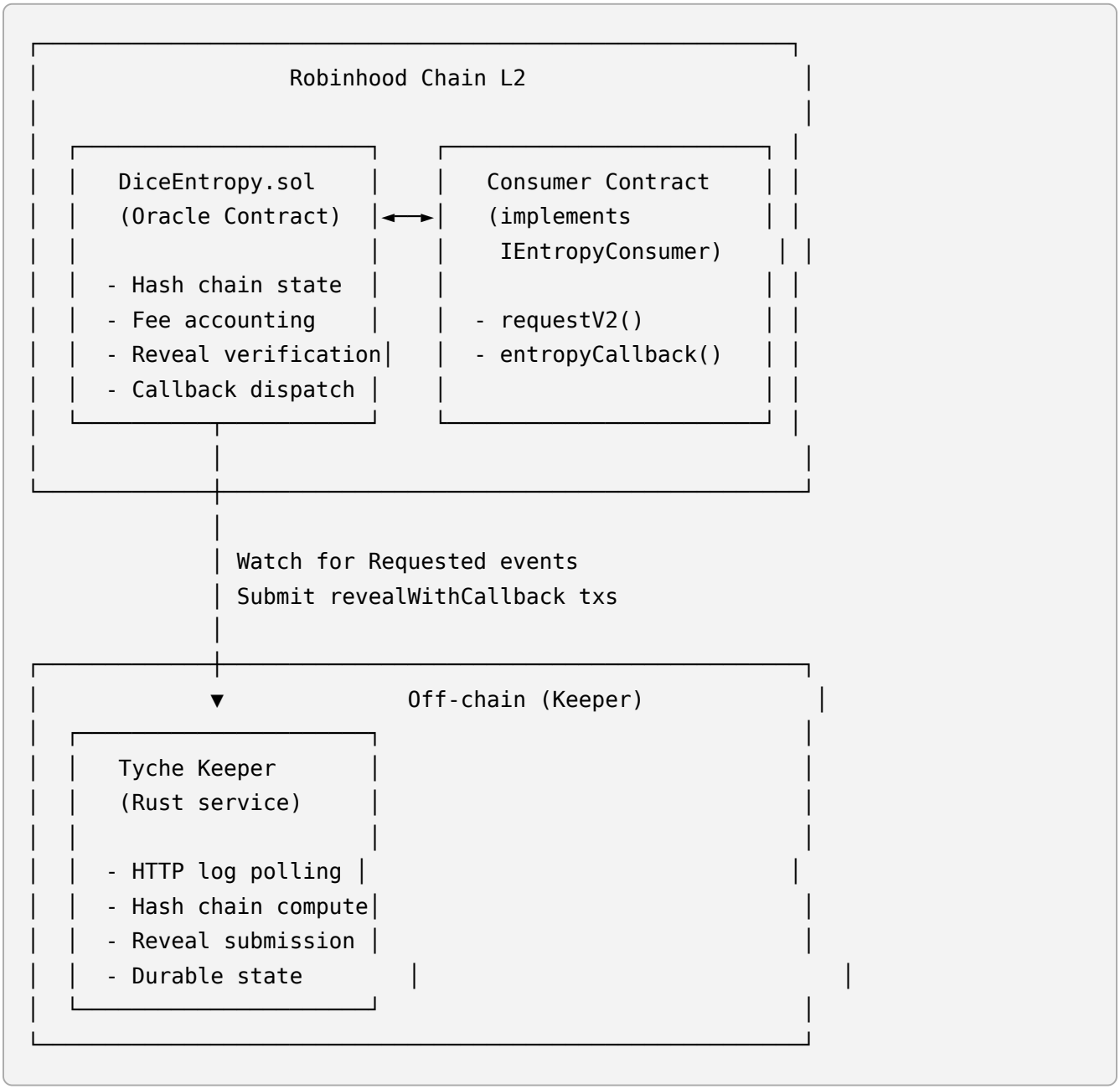
- A **provider** generates a hash chain of random values (live v10 registration length is 500,000; longer chains are supported) by repeatedly hashing a secret seed with Keccak256.
- The provider commits the **root** (final hash) of this chain to the onchain contract during deployment.
- When a user requests randomness, they contribute their own random value.
- The provider reveals the next hash in the chain, which the contract verifies by hashing it and checking against the committed root.
- The final random number is `Keccak256(userRandomness || providerContribution || bytes32(0))`, combining both parties' inputs with the fixed zero blockhash argument used by `requestV2`.

This approach provides: - **Provider cannot precompute a completed result**: The provider does not know the user's random value at request time. - **User cannot choose the provider contribution**: The user's value is committed before the provider reveals. - **Liveness remains provider-dependent**: The provider can still withhold a reveal. Eligible requests become refundable after the configured delay. - **Verifiable onchain**: Anyone can verify a completed reveal by hashing it. - **Agent-friendly**: Immutable contract code, deterministic fees, and documented verification make Dice Protocol straightforward for AI agents and automation systems to integrate. Agents still need key security, transaction simulation, receipt checks, timeout handling, and explicit refund logic. A complete SKILL.md is published for agent consumption.

---

2. Architecture

2.1 System Components



2.2 DiceEntropy Contract

The onchain contract is immutable (no proxy pattern) and handles:

- **Provider registration**: The constructor auto-registers the default provider with its hash chain commitment, chain length, and metadata.
- **Request handling**: Users call `requestV2()` with a user random value and optional gas limit. The contract assigns a sequence number and emits a `Requested` event.
- **Reveal verification**: When the keeper calls `revealWithCallback()`, the contract verifies the provider's contribution by hashing it and comparing to the stored commitment. If valid, it advances the commitment pointer.
- **Callback dispatch**: After verification, the contract calls the consumer's `entropyCallback()` function with the combined random number, using an `excessivelySafeCall` pattern with a configurable gas limit.
- **Fee accounting**: Each request charges a flat fee (0.000025 ETH). Fees accrue in the contract and are withdrawable to a designated vault address by the admin.

2.3 Tyche Keeper

Tyche is a Rust-based off-chain service that:

1. **Monitors the blockchain** for `Requested` events via HTTP polling (1-second interval) using `eth_getLogs`.
2. **Computes the reveal value** for each request by traversing the precomputed hash chain to the correct sequence number.
3. **Submits `revealWithCallback` transactions** from the keeper wallet.
4. **Maintains state** in a database (request tracking, last-processed block, etc.).
5. **Uses monitored, recoverable operations** so request processing can resume after transient failures.

2.4 Wallet Separation

Dice Protocol separates administrative, fee-recipient, and reveal-submission roles:

Role	Address	Key Type	Purpose
Admin	`0x4ACD...`	Cold	Contract admin (fee changes, withdrawals, provider management). Accepted admin role on-chain.
Vault	`0x918E...`	Cold	Fee recipient (receives withdrawn fees)
Keeper / provider	`0x8741...`	Hot	Submits reveals and may update provider-specific URI, maximum-hash, and callback-gas settings; cannot change protocol fee/admin or withdraw protocol fees

The keeper wallet only holds enough ETH for gas. It cannot withdraw protocol fees or change admin-controlled protocol parameters. It can update provider-specific settings listed above. If compromised, an attacker could affect reveal timing/liveness and provider-specific configuration, but cannot withdraw protocol fees.

3. Cryptographic Design

3.1 Hash Chain Construction

The hash chain is an S/KEY-like construction using Keccak256:

1. Start with a 32-byte secret seed `s`.
2. Compute `h₁ = Keccak256(s)`.
3. For `i = 2..N`: compute `hᵢ = Keccak256(hᵢ₋₁)`.
4. The chain is `[h₁, h₂, ..., h_N]` where `h_N` is the **root commitment**.
5. Reveal order is **reverse**: `h_N` is revealed first (for sequence 0), then `h_{N-1}` (sequence 1), etc.

Verification: `Keccak256(h_{N-k}) == h_{N-k+1}` (the previously revealed value). Each reveal hashes to the previous commitment, proving chain membership without revealing future values.

3.2 Random Number Combination

The final random number combines two sources:

```
randomNumber = Keccak256(userRandomness || providerContribution || bytes32(0))
```

- **userRandomness**: Provided by the requester at request time. Unknown to the provider at commitment time.
- **providerContribution**: The hash chain value at the current sequence number. Determined at chain generation time, unknown to the user at request time.
- **Fixed third input**: `requestV2` sets `useBlockhash = false`, so the contract hashes `bytes32(0)` as the third input. Generate the user contribution locally with a cryptographically secure random number generator.

For a completed reveal, neither requester nor provider can unilaterally choose the output. A provider can still withhold service; eligible requests become refundable after the configured delay.

3.3 Commitment Metadata

The provider stores non-secret rotation metadata onchain. The live metadata identifies the chain generation context and registered length, but it does not publish the raw hash-chain secret or future preimages.

The keeper reconstructs the chain using private provider configuration plus the public chain, provider, and contract context. Publishing the raw secret would make future provider contributions predictable, so it must remain offchain and private.

3.4 Security Properties

Property	Guarantee
Unpredictability	Provider cannot predict user's random value at commitment time
Completed-result control	Neither party unilaterally chooses a completed result. The provider can still withhold a reveal; refunds mitigate fee loss.
Verifiability	Each reveal is verifiable onchain via Keccak256
Tamper resistance	Immutable contract, no upgrade path
Gas bounded	Effective callback gas is `max(requested, provider.defaultGasLimit)`, rounded up to 10,000, then bounded by `MAX_GAS_LIMIT`. If `defaultGasLimit` is 0, stored callback gas is 0 and the gas-limited path does not run. Live default is 200,000.
Chain exhaustion protection	`OutOfRandomness` error when chain depleted

3.5 Zero Reveal Delay

Some commit-reveal oracles enforce a mandatory waiting period between request and reveal — typically 10 to 20 blocks. Dice Protocol does not. This section explains why zero delay is not just acceptable but architecturally correct for this deployment.

The purpose of a reveal delay is to protect a third entropy source: the block hash. On Ethereum L1, block hashes are produced through competitive mining — no single party controls them, making them a reasonably unbiased input. The delay ensures this value is finalized before the provider reveals, preventing the sequencer from cherry-picking a favorable block.

Dice Protocol does not incorporate block data into its randomness. The output is purely two-party: the user's contribution and the provider's hash chain reveal. Since block hash is not part of the equation, there is nothing for a delay to protect. Adding one would impose latency without adding security — the hash chain values are fixed at registration time and are immune to reordering or reorgs.

This is specific to the L2 context. Robinhood Chain runs on a single centralized sequencer. Block production is deterministic and ordered — there is no competitive mining process that would make block hashes independently unpredictable. Treating an L2 block hash as an entropy source would import a value controlled by one operator, which is the opposite of the decentralization that makes block hashes useful on L1.

The tradeoff is clear: zero delay gives ~1–3 second typical request-to-reveal latency. For a completed reveal, neither party unilaterally chooses the output under the stated model. The provider can still withhold a reveal and affect liveness; the refund path mitigates fee loss but does not produce a random result.

---

4. Economic Model

4.1 Fee Structure

Dice Protocol uses a **single flat fee** model:

- **Fee per request**: 0.000025 ETH (25,000,000,000,000 wei)
- **Fee destination**: 100% to the contract's accrued fee pool
- **Withdrawal**: Admin calls `withdrawFees()`, sending the full balance to the vault address

There is no provider-specific fee split. The exact request fee accrues to the protocol fee pool and is withdrawable to the vault. Reveal gas and operating costs vary with network conditions and consumer callback behavior, so no fixed per-request margin is claimed.

4.2 Gas Economics

Gas costs measured from actual mainnet transactions (Robinhood Chain, ~0.095 gwei):

Operation	Gas Cost	ETH Cost (at ~0.095 gwei)
Reveal (keeper)	~53,506	~0.0000051
Callback (included in reveal tx)	~included above	included
Total per reveal	**~53,506**	**~0.0000051**

Gas cost varies with network conditions and callback behavior. Request fees and transaction gas are separate; do not infer a fixed margin from a single measurement.

4.3 Chain Renewal

Each hash chain has a finite configured length. Live v10 currently has 500,000 entries registered. At current usage projections (100 requests/day), a chain lasts ~5,000 days. When the chain is exhausted, the admin registers a new commitment via `registerFor()` and the keeper is configured with the new seed.

---

5. Smart Contract API

5.1 Core Functions

`requestV2(address provider, bytes32 userRandomNumber, uint32 gasLimit) → uint64`

Request randomness from a specific provider. The caller must send ETH equal to the provider's fee.

- `provider`: The provider address (must be registered)
- `userRandomNumber`: User's random contribution (32 bytes)

- `gasLimit`: Requested callback gas. Effective limit is `max(requested, provider.defaultGasLimit)`, rounded up to 10,000 units, and must not exceed `MAX_GAS_LIMIT` (655,350,000). A 0 request with live default 200,000 becomes 200,000. If `defaultGasLimit` is 0, stored callback gas is 0 regardless of the request.
- Returns: Assigned sequence number

`revealWithCallback(address provider, uint64 sequenceNumber, bytes32 userContribution, bytes32 providerContribution)`

Called by the keeper to reveal the provider's contribution and trigger the consumer's callback.

- Verifies `Keccak256(providerContribution) == currentCommitment`
- Computes `randomNumber = Keccak256(userContribution || providerContribution || bytes32(0))`
- Calls `entropyCallback(sequenceNumber, provider, randomNumber)` on the requester
- Uses `excessivelySafeCall` with the stored effective limit: `max(requested, provider.defaultGasLimit)`, rounded up to 10,000. If stored callback gas is 0, the callback uses remaining gas.

`getFee(address provider) → uint128`

Returns the current fee for a provider.

`withdrawFees(uint128 amount)`

Admin-only. Withdraws accrued fees to the vault address.

`setFee(uint128 feeInWei)`

Admin-only. Updates the request fee.

5.2 Consumer Interface

Consuming contracts must implement `IEntropyConsumer`:

```
interface IEntropyConsumer {
    function entropyCallback(
        uint64 sequenceNumber,
        address provider,
        bytes32 randomNumber
    ) internal;

    function getEntropy() internal view returns (address);
}
```

5.3 Events

Event	Description
`Requested(provider, caller, sequenceNumber, userContribution, gasLimit)`	Emitted when a request is made
`Revealed(provider, caller, sequenceNumber, randomNumber, ...)`	Emitted when a reveal completes
`Registered(provider, ...)`	Emitted when a provider is registered

---

6. Tyche Keeper

6.1 Operation

Tyche operates as a continuous block-watching service:

1. **Initialization**: Reads public provider info from the contract and reconstructs the hash chain in memory from private provider configuration plus public chain, provider, and contract context. Onchain metadata is not the raw hash-chain secret.
2. **Block watching**: HTTP polling at 1-second intervals using `eth_blockNumber` + `eth_getLogs`. The keeper fetches the latest block and queries event logs in 9-block batches. No WebSocket dependency — pure HTTP for maximum reliability across RPC providers.
3. **Reveal computation**: For each request, computes the reveal value by indexing into the precomputed hash chain at the sequence number offset.
4. **Transaction submission**: Sends `revealWithCallback` transactions from the keeper wallet with appropriate gas.
5. **State persistence**: Records request progress and outcomes for recovery and observability.

6.2 Operational configuration

Keeper configuration, credentials, RPC topology, and persistence details are private operational information. Integrators depend only on the public contract, events, SDK, and status surface.

6.3 Deployment

Operational deployment details are private. Public reliability and current health are reported through the protocol status surface. - Graceful shutdown via SIGINT - Memory footprint: ~8.4 MB

---

7. Security Analysis

7.1 Attack Vectors

Attack	Mitigation
Provider withholds reveal	No onchain penalty in v1; relies on service uptime. Future: slashing.
User front-runs reveal	User commits random value before provider reveals; provider cannot see it in advance.
Callback gas behavior	Effective callback gas is `max(requested, provider.defaultGasLimit)`, rounded up to 10,000, rejected above `MAX_GAS_LIMIT`. Live default 200,000: a 0 request becomes 200,000; an explicit 500,000 request is honored. If `defaultGasLimit` is 0, stored gas is 0. Uses `excessivelySafeCall`.
Chain exhaustion	`OutOfRandomness` revert when sequence exceeds chain length.
Private key compromise	Three-wallet separation limits blast radius. Keeper cannot steal fees.
Contract reentrancy	`excessivelySafeCall` pattern prevents callback reentrancy into reveal logic.

7.2 Audit Results

The contract has been reviewed using automated analysis tools (Slither, Aderyn) by the founding team. No third-party security audit has been conducted.

Severity	Count	Status
Critical	0	—
High	0	—
Medium	2	M1 (gas griefing) mitigated, M2 (block.timestamp PRNG) acceptable
Low	2	L1 (excess fees) intentional, L2 (param naming) cosmetic
Info	3	Acknowledged

Note: The automated analysis above covers the Solidity contract only. A separate operational review identified a keeper credential exposure incident (see security-audit.md). This was an operational issue, not a contract vulnerability. The contract code itself has 0 critical and 0 high findings from automated analysis.

7.3 Immutability

The contract has no proxy, no upgrade mechanism, and no governance. Once deployed, the logic cannot be modified. Parameters that can be changed by admin: - `setFee()` : Adjust the request fee - `setDefaultProvider()` : Change the default provider - `withdrawFees()` : Withdraw accrued fees - `registerFor()` : Register a new provider

---

8. Integration Guide

8.1 TypeScript SDK

```
import { DiceProtocol } from '@diceprotocol/sdk';

const dice = new DiceProtocol({
  rpcUrl: 'https://rpc.mainnet.chain.robinhood.com',
  contractAddress: '0xd8a0680e7699526b57140ed4eafdcc7219dc0a0c',
});

// Request randomness
const seqNum = await dice.requestRandom(signer, undefined, userRandom, 200000);

// The consumer contract receives the result through entropyCallback
```

8.2 Solidity Integration

```
import { IEntropyConsumer } from
"@diceprotocol/sdk/solidity/IEntropyConsumer.sol";
import { IEntropy } from "@diceprotocol/sdk/solidity/IEntropy.sol";

contract MyGame is IEntropyConsumer {
    IEntropy public immutable dice;
    address public provider;

    constructor(address _dice, address _provider) {
        dice = IEntropy(_dice);
        provider = _provider;
    }

    function roll(bytes32 userRandom) external payable {
        // Generate userRandom client-side with a CSPRNG.
        dice.requestV2{value: msg.value}(provider, userRandom, 200000);
    }

    function entropyCallback(uint64 seq, address, bytes32 random) internal
override {
        // Use random number
        uint256 result = uint256(random) % 6 + 1; // Dice roll 1-6
    }

    function getEntropy() internal view override returns (address) {
        return address(dice);
    }
}
```

---

9. Roadmap

v1.0 (Current — July 2026)

- ✅ DiceEntropy contract deployed on Robinhood Chain mainnet
- ✅ Tyche keeper operational with auto-reveal (~1–3s typical latency)
- ✅ TypeScript SDK published to npm (`@diceprotocol/sdk`)
- ✅ Fee set to 0.000025 ETH (live on-chain)
- ✅ Admin role accepted (0x4ACD...)
- ✅ Automated security analysis (Slither + Aderyn) — 0 critical/0 high
- ✅ Contract source published and verified on Blockscout
- ⬜ Third-party security audit (pending)

v1.1 (Q3 2026)

- Multi-provider support (redundancy)
- Tyche monitoring dashboard
- Self-hosted RPC node for improved latency

v2.0 (Q4 2026)

- Slashing mechanism for provider non-responsiveness (requires new deployment)
- Variable fee tiers (basic/premium)
- Cross-chain expansion to other Nitro L2s

---

10. Appendix

A. Contract Addresses

Component	Address
DiceEntropy	`0xd8a0680e7699526b57140ed4eafdcc7219dc0a0c`
TestConsumer	`0xa1d2C96EC9E5110f962264C5489D78299a88C677`
Admin	`0x4ACD2C88a239a924E47Fc4995114ca1Bb0CA3CaD`
Vault	`0x918EAF0b2589710B0D85ef48C12a343E68263841`
Keeper	`0x8741b8a825644D9Ef18Faf2DAB5e9b47B900F2b6`

B. Network Configuration

Parameter	Value
Chain ID	4663
RPC URL	`https://rpc.mainnet.chain.robinhood.com`
Explorer	`https://robinhoodchain.blockscout.com`
Fee	25,000,000,000,000 wei (0.000025 ETH)
Hash chain length	500,000 (live v10 registration)
defaultGasLimit	200,000

C. Hash Chain Parameters

Parameter	Value
Algorithm	Keccak256
Registered span	500,000 values (sequences 3 through 500002; end sequence 500003)
Original commitment	`0xb533ab9c0451c5b58c42ca93a2ad5fceeba8b07fb4ea95089ff6474def4762b3` (registered at sequence 3; query `getProviderInfoV2` for the advancing current commitment)
Sample interval	1 (every hash stored)

D. References

- S/KEY one-time password system (RFC 1760) — hash chain construction
- Arbitrum Nitro — L2 architecture context
- Keccak256 (NIST FIPS 202) — hash function

Dice Protocol is developed and maintained as independent infrastructure for Robinhood Chain. Architecture adapted from Pyth Entropy (Apache-2.0).

Refunds (v10)

If a request is not revealed within about 60–90 seconds, the original requester can reclaim the exact fee:

```
// refundDelayBlocks = 6 on Robinhood Chain (L1 blocks ≈ 12s)
dice.refundRequest(provider, sequenceNumber);
```

Notes: - Only the original requester can refund - Request must still be active (not revealed / settled) - Delay is L1-block based because Robinhood/Arbitrum Nitro uses L1 `block.number` - Live contract: `0xd8a0680e7699526b57140ed4eafdcc7219dc0a0c` - Live fee: exact `0.000025 ETH`

Failed-request economics

A refund returns the exact request fee, but transaction gas is not refunded:

- The requester pays gas for the original request and for `refundRequest`.
- Dice retains no request fee and earns nothing from an unrevealed, refunded request.
- If no reveal transaction was submitted, Dice has no direct reveal-transaction gas cost for that request.
- If a reveal transaction was submitted but reverted, Dice bears that failed transaction's gas cost.