BLOCKSEC

Security Audit
Report for Morph Reth

Date: June 23, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Morph
Target	Morph Reth

Version History

Version	Date	Description
1.0	June 23, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-06-24 08:49:08+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of Morph Reth of Morph.

Morph Reth is a Rust-based Morph L2 execution client built on top of Reth and Revm. It implements Morph-specific execution semantics, including Morph transaction types, ERC20 gas token payment, L1 data fee accounting, FeeVault routing, L1 message execution, custom precompile sets, and Engine API block assembly/import flows. The project aims to remain compatible with Morph's Go implementation while leveraging the Reth stack for modular execution, payload building, transaction pool management, and RPC handling.

Note this audit only focuses on the smart contracts in the following directories/files:

- crates/revm/src/precompiles.rs
- crates/revm/src/evm.rs
- crates/revm/src/handler.rs
- crates/primitives/src/transaction/morph_transaction.rs
- crates/primitives/src/transaction/envelope.rs
- crates/txpool/src/morph_tx_validation.rs
- crates/txpool/src/maintain.rs
- crates/rpc/src/eth/call.rs
- crates/rpc/src/eth/transaction.rs
- crates/engine-api/src/builder.rs

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
Morph Reth	`Version 1`	`ac2e035123f237e88a39b60cb9669168bd2bfdf8`
	`Version 2`	`f60c2b2bbde1aaee7ab02244c96a70a55e7aaf76`

[1] https://github.com/morph-l2/morph-reth

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation

- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **thirteen** potential security issues. Besides, we have **three** recommendations and **two** notes.

- High Risk: 4
- Low Risk: 9
- Recommendation: 3
- Note: 2

ID	Severity	Description	Category	Status
1	High	Potential chain state inconsistency due to incorrect `MorphTx` version detection logic	Security Issue	Fixed
2	High	Potential chain state inconsistency due to incorrect fee validation logic	Security Issue	Fixed
3	High	Potential transaction version inconsistency due to incorrect `MorphTx` version selection logic	Security Issue	Fixed
4	High	Potential transaction type inconsistency due to incorrect `gas_price` handling in `MorphTx` construction	Security Issue	Fixed
5	Low	Inconsistent rounding direction for token fee gas estimation	Security Issue	Confirmed
6	Low	Lack of trie backend configuration check in block assembly	Security Issue	Confirmed
7	Low	Improper logic in function `validate_version()`	Security Issue	Fixed
8	Low	Lack of `gas_limit` check in function `maintain_morph_pool()`	Security Issue	Fixed
9	Low	Non-Atomic canonical head lookup in function `current_head()`	Security Issue	Fixed
10	Low	Underestimated token fee in `MorphTx` pool validation	Security Issue	Fixed
11	Low	Lack of default `chain_id` handling in `MorphTx` RPC construction	Security Issue	Fixed
12	Low	Lack of RPC input validation in `MorphTx` construction	Security Issue	Fixed
13	Low	Inconsistent gas estimation behavior between Rust and Go clients implementation	Security Issue	Confirmed
14	-	Update the precompile set comment reference	Recommendation	Fixed
15	-	Apply `MorphTx` validation in the RPC simulation path	Recommendation	Confirmed

16	-	Handle database read errors in storage `original_value` restoration	Recommendation	Confirmed
17	-	Trusted token and oracle assumptions	Note	-
18	-	Trusted upstream `Revm` baseline	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Potential chain state inconsistency due to incorrect `MorphTx` version detection logic

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `morph_transaction.rs`, the function `decode_fields()` determines the `MorphTx` version by inspecting the first byte of the payload. Payloads whose first byte is an RLP list prefix (`0xC0` or greater) are routed to the `V0` decoder, while payloads starting with version byte `0x01` are routed to the `V1` decoder.

However, the Rust client implementation does not treat a leading zero byte (`0x00`) as a `V0` payload indicator. On the contrary, in the Go client implementation, both a leading zero byte and an RLP list prefix are routed to the `V0` decoder.

As a result, a `MorphTx V0` payload with a leading `0x00` may be accepted by the Go client implementation but rejected by the Rust client implementation as an unsupported version. If such a transaction is included in a block, Rust and Go nodes may disagree on block validity, leading to a potential chain state inconsistency.

```
340   /// Decodes the inner fields from RLP bytes (after txType byte is consumed).
341   ///
342   /// Version detection based on first byte:
343   /// - V0 format: first byte is 0 or RLP list prefix (>= 0xC0) →direct RLP decode
344   /// - V1+ format: first byte is version (0x01, 0x02, ...) →skip version byte, then RLP decode
345   ///
346   /// V0 RLP: ChainID, Nonce, GasTipCap, GasFeeCap, Gas, To, Value, Data, AccessList, FeeTokenID,
          FeeLimit
347   /// V1 RLP: ChainID, Nonce, GasTipCap, GasFeeCap, Gas, To, Value, Data, AccessList, FeeTokenID,
          FeeLimit, Reference, Memo
348   pub fn decode_fields(buf: &mut &[u8]) -> alloy_rlp::Result<Self> {
349       if buf.is_empty() {
350           return Err(alloy_rlp::Error::InputTooShort);
351       }
352
353       let first_byte = buf[0];
354
355       // Check first byte to determine version:
```

```
356      // - V0 format (legacy AltFeeTx): first byte is RLP list prefix (0xC0-0xFF), no version
             prefix
357      // - V1+ format: first byte is version (0x01, 0x02, ...) followed by RLP
358      if first_byte >= 0xC0 {
359          // V0 format: direct RLP decode (legacy compatible)
360          Self::decode_fields_v0(buf)
361      } else if first_byte == MORPH_TX_VERSION_1 {
362          // V1 format: first byte is version, rest is RLP
363          // Skip the version byte
364          *buf = &buf[1..];
365          Self::decode_fields_v1(buf)
366      } else {
367          Err(alloy_rlp::Error::Custom("unsupported morph tx version"))
368      }
369  }
```

Listing 2.1: morph-reth/crates/primitives/src/transaction/morph_transaction.rs

Impact `MorphTx V0` payloads with a leading zero byte may be accepted by the Go client implementation but rejected by the Rust client implementation. If such transactions are propagated or included in a block, Rust nodes may reject transactions or blocks that Go nodes consider valid.

Suggestion Add the zero-byte case to the `V0` routing branch in function `decode_fields()` to match the documented behavior.

2.1.2 Potential chain state inconsistency due to incorrect fee validation logic

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `handler.rs`, the function `validate_env()` performs `EIP-1559` fee parameter validation for `MorphTx` transactions only when the transaction pays fees in the native token. This validation path is not applied to token-fee `MorphTx` transactions. In the Go client implementation, `EIP-1559` fee parameter validation, including the requirement that `max_fee_per_gas >= base_fee`, is applied to all non-L1-message transactions regardless of the fee payment method.

As a result, a token-fee `MorphTx` with `max_fee_per_gas` below the current `base_fee` can pass validation in Rust client implementation but be rejected by the Go client implementation. If such a transaction is included in a block, Rust and Go nodes may disagree on block validity.

```
219  fn validate_env(&self, evm: &mut Self::Evm) -> Result<(), Self::Error> {
220      // For L1 message transactions
221      if evm.ctx_ref().tx().is_l1_msg() {
222          // L1 messages have zero gas price, so skip gas price validation
223          return Ok(());
224      }
225
226      // Standard validation.
227      // Note: revm maps MorphTx (type 0x7F) to `TransactionType::Custom`,
```

```
228         // which skips gas-price checks entirely.
229         validation::validate_env::<_, Self::Error>(evm.ctx())?;
230
231         // MorphTx V1 ETH-fee: enforce the EIP-1559 priority-fee rule.
232         // Token-fee MorphTx skips it (fees paid in ERC20); simulation
233         // paths also skip it.
234         if evm.ctx_ref().tx().is_morph_tx()
235             && !evm.ctx_ref().tx().uses_token_fee()
236             && !evm.ctx_ref().cfg().is_fee_charge_disabled()
237         {
238             let base_fee = Some(evm.ctx_ref().block().basefee() as u128);
239             validation::validate_priority_fee_tx(
240                 evm.ctx_ref().tx().max_fee_per_gas(),
241                 evm.ctx_ref()
242                     .tx()
243                     .max_priority_fee_per_gas()
244                     .unwrap_or_default(),
245                 base_fee,
246                 evm.ctx_ref().cfg().is_priority_fee_check_disabled(),
247             )?;
248         }
249
250         Ok(())
251     }
```

Listing 2.2: morph-reth/crates/revm/src/handler.rs

Impact The divergent validation can lead to state inconsistency between Rust and Go nodes, as one accepts a block containing such a transaction while the other rejects it.

Suggestion Apply the `EIP-1559` fee parameter check to all `MorphTx` transactions regardless of the fee payment token.

2.1.3 Potential transaction version inconsistency due to incorrect `MorphTx` version selection logic

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `transaction.rs`, the function `try_build_morph_tx_from_request()` assigns version `V1` to all `MorphTx` transactions constructed from RPC requests. In the Go client implementation, the `MorphTx` version is selected based on the request fields. If `V1` specific fields, such as a non-zero `reference` or a non-empty `memo`, are present, the transaction is constructed as `V1`. Otherwise, a `MorphTx` with only `fee_token_id` set is constructed as `V0`.

As a result, the same RPC request with `fee_token_id` set but without reference or memo may produce a `V1` transaction in Rust client implementation and a `V0` transaction in Go client implementation, which is incorrect.

```
230     fn try_build_morph_tx_from_request(
231         req: &alloy_rpc_types_eth::TransactionRequest,
```

```
232         fee_token_id: U64,
233         fee_limit: U256,
234         reference: Option<alloy_primitives::B256>,
235         memo: Option<alloy_primitives::Bytes>,
236     ) -> Result<Option<TxMorph>, &'static str> {
237         let fee_token_id_u16 = u16::try_from(fee_token_id.to::<u64>()).map_err(|_| "invalid token")
                ?;
238
239         // Check if this should be a MorphTx
240         let has_fee_token = fee_token_id_u16 > 0;
241         let has_reference = reference.is_some();
242         let has_memo = memo.as_ref().is_some_and(|m| !m.is_empty());
243
244         if !has_fee_token && !has_reference && !has_memo {
245             // No Morph-specific fields →standard Ethereum tx
246             return Ok(None);
247         }
248
249         // All MorphTx are constructed as Version 1
250         let version = morph_primitives::transaction::morph_transaction::MORPH_TX_VERSION_1;
251
252         // Now build the MorphTx
253         let chain_id = req
254             .chain_id
255             .ok_or("missing chain_id for morph transaction")?;
256         let gas_limit = req.gas.unwrap_or_default();
257         let nonce = req.nonce.unwrap_or_default();
258         let max_fee_per_gas = req.max_fee_per_gas.or(req.gas_price).unwrap_or_default();
259         let max_priority_fee_per_gas = req.max_priority_fee_per_gas.unwrap_or_default();
260         let access_list: AccessList = req.access_list.clone().unwrap_or_default();
261         let input = req.input.clone().into_input().unwrap_or_default();
262         let to = req.to.unwrap_or(TxKind::Create);
263
264         let morph_tx = TxMorph {
265             chain_id,
266             nonce,
267             gas_limit,
268             max_fee_per_gas,
269             max_priority_fee_per_gas,
270             to,
271             value: req.value.unwrap_or_default(),
272             access_list,
273             input,
274             fee_token_id: fee_token_id_u16,
275             fee_limit,
276             version,
277             reference,
278             memo,
279         };
280
281         // Validate all MorphTx constraints: version-specific rules, gas fee ordering,
282         // and memo length. This catches invalid combinations early at the RPC layer.
283         morph_tx.validate()?;
```

```
284
285         Ok(Some(morph_tx))
286     }
```

Listing 2.3: morph-reth/crates/rpc/src/eth/transaction.rs

Impact The same RPC request can produce different `MorphTx` versions across clients, leading to inconsistent transaction encoding and signature computation.

Suggestion Align Rust `MorphTx` version selection with the Go client implementation.

2.1.4 Potential transaction type inconsistency due to incorrect `gas_price` handling in `MorphTx` construction

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `transaction.rs`, function `try_build_morph_tx_from_request()` treats `gas_price` as a fallback for `max_fee_per_gas` when constructing a `MorphTx`. If a request contains both `gas_price` and `MorphTx` specific fields, the Rust client implementation still constructs a `MorphTx`. In the Go client implementation, the presence of `gas_price` forces the transaction type to `LegacyTxType`, unconditionally overriding any `MorphTx` specific field detection. A request with both `gas_price` and `Morph` specific fields therefore produces a `MorphTx` in Rust client implementation but a legacy transaction in Go client implementation, which is incorrect.

```
230     fn try_build_morph_tx_from_request(
231         req: &alloy_rpc_types_eth::TransactionRequest,
232         fee_token_id: U64,
233         fee_limit: U256,
234         reference: Option<alloy_primitives::B256>,
235         memo: Option<alloy_primitives::Bytes>,
236     ) -> Result<Option<TxMorph>, &'static str> {
237         let fee_token_id_u16 = u16::try_from(fee_token_id.to::<u64>()).map_err(|_| "invalid token")
                ?;
238
239         // Check if this should be a MorphTx
240         let has_fee_token = fee_token_id_u16 > 0;
241         let has_reference = reference.is_some();
242         let has_memo = memo.as_ref().is_some_and(|m| !m.is_empty());
243
244         if !has_fee_token && !has_reference && !has_memo {
245             // No Morph-specific fields →standard Ethereum tx
246             return Ok(None);
247         }
248
249         // All MorphTx are constructed as Version 1
250         let version = morph_primitives::transaction::morph_transaction::MORPH_TX_VERSION_1;
251
252         // Now build the MorphTx
253         let chain_id = req
```

```
254             .chain_id
255             .ok_or("missing chain_id for morph transaction")?;
256         let gas_limit = req.gas.unwrap_or_default();
257         let nonce = req.nonce.unwrap_or_default();
258         let max_fee_per_gas = req.max_fee_per_gas.or(req.gas_price).unwrap_or_default();
259         let max_priority_fee_per_gas = req.max_priority_fee_per_gas.unwrap_or_default();
260         let access_list: AccessList = req.access_list.clone().unwrap_or_default();
261         let input = req.input.clone().into_input().unwrap_or_default();
262         let to = req.to.unwrap_or(TxKind::Create);
263
264         let morph_tx = TxMorph {
265             chain_id,
266             nonce,
267             gas_limit,
268             max_fee_per_gas,
269             max_priority_fee_per_gas,
270             to,
271             value: req.value.unwrap_or_default(),
272             access_list,
273             input,
274             fee_token_id: fee_token_id_u16,
275             fee_limit,
276             version,
277             reference,
278             memo,
279         };
280
281         // Validate all MorphTx constraints: version-specific rules, gas fee ordering,
282         // and memo length. This catches invalid combinations early at the RPC layer.
283         morph_tx.validate()?;
284
285         Ok(Some(morph_tx))
286     }
```

Listing 2.4: morph-reth/crates/rpc/src/eth/transaction.rs

Impact The same RPC request can result in different transaction types between clients, affecting fee payment semantics and transaction processing behavior.

Suggestion Align the Rust and Go transaction type selection logic.

2.1.5 Inconsistent rounding direction for token fee gas estimation

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In file `call.rs`, function `token_amount_to_eth()` converts the user's `ERC20` token balance into an equivalent native token amount for gas allowance calculation in functions `eth_estimateGas()` and `eth_call()`. This conversion uses floor division. In the Go client implementation, the function `AltToEth()` performs ceiling division by adding one whenever a non-zero remainder exists. For the same token balance and exchange rate, the Rust client

implementation therefore computes a lower ETH-equivalent gas allowance than Go client implementation.

```
311   fn token_amount_to_eth(token_amount: U256, info: &TokenFeeInfo) -> Option<U256> {
312       if info.price_ratio.is_zero() || info.scale.is_zero() {
313           return None;
314       }
315       Some(token_amount.saturating_mul(info.price_ratio) / info.scale)
316   }
```

Listing 2.5: morph-reth/crates/rpc/src/eth/call.rs

Impact When the token fee conversion produces a remainder, Rust client implementation may calculate a lower gas allowance than the Go client implementation. As a result, a transaction may fail gas estimation in Rust client implementation while succeeding in Go client implementation.

Suggestion Align token fee rounding direction with the Go client implementation by using ceiling division.

Feedback from the project The project clarifies that Rust client's floor rounding is intentional and represents the safe, internally consistent behavior. The ceil rounding used by Go client in the token to ETH conversion path is therefore treated as an implementation difference rather than a defect in Rust client.

2.1.6 Lack of trie backend configuration check in block assembly

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the Go client implementation, the function `AssembleL2Block()` checks whether the `Jade` fork status is consistent with the configured trie backend `UseZktrie`. If the configuration indicates an incompatible fork/backend combination, block assembly is rejected. This prevents a node from producing blocks under an invalid `zktrie` to `MPT` migration configuration.

The function `assemble_l2_block()` in Rust client implementation does not perform an equivalent configuration consistency check. As a result, Rust client implementation may proceed with block assembly without explicitly validating the fork/backend compatibility enforced by the Go client implementation.

```
169   async fn assemble_l2_block(
170       &self,
171       params: AssembleL2BlockParams,
172   ) -> EngineApiResult<ExecutableL2Data> {
173       let started = Instant::now();
174       let result = self.build_l2_payload(params, None, None).await;
175       self.metrics
176           .assemble_l2_block_duration_seconds
177           .record(started.elapsed());
178
179       let built_payload = result.inspect_err(|_| {
```

```
180             self.metrics.assemble_l2_block_failures_total.increment(1);
181         })?;
182         let executable_data = built_payload.executable_data;
183
184         tracing::debug!(
185             target: "morph::engine",
186             block_hash = %executable_data.hash,
187             gas_used = executable_data.gas_used,
188             tx_count = executable_data.transactions.len(),
189             "L2 block assembled successfully"
190         );
191
192         Ok(executable_data)
193     }
```

Listing 2.6: morph-reth/crates/engine-api/src/builder.rs

Impact In the Go client implementation, a node with an incompatible `Jade` fork and trie backend configuration is rejected during block assembly. Rust client implementation does not perform an equivalent safeguard, which may allow block assembly to proceed without the same fork/backend compatibility check.

Suggestion Evaluate whether an equivalent fork-aware configuration check is warranted in the block assembly path.

Feedback from the project The project clarifies that the Rust client does not expose a trie backend switch equivalent to the Go client's `UseZktrie` option. In the Rust client, `pre-Jade` and `post-Jade` state root enforcement is handled centrally by the function `state_root_enforced_at()`. Therefore, the corresponding Go client check is tied to its own trie backend configuration model and does not directly apply to the Rust client.

2.1.7 Improper logic in function `validate_version()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `morph_transaction.rs`, the function `validate_version()` rejects any `MorphTx V0` transaction where the `reference` field is `Some`, regardless of the underlying value. In the Go client implementation, function `ValidateMorphTxVersion()` allows a `MorphTx V0` to carry a `reference` field as long as the `reference` value is all zeros. A `V0` transaction containing a zero-valued `reference` is therefore accepted by the Go client implementation, but rejected by the Rust client implementation.

```
204     pub fn validate_version(&self) -> Result<(), &'static str> {
205         match self.version {
206             MORPH_TX_VERSION_0 => {
207                 // Version 0 requires FeeTokenID > 0 (legacy format used for alt-fee transactions)
208                 if self.fee_token_id == 0 {
209                     return Err("version 0 MorphTx requires FeeTokenID > 0");
210                 }
```

```
211             // Version 0 does not support Reference field
212             if self.reference.is_some() {
213                 return Err("version 0 MorphTx does not support Reference field");
214             }
215             // Version 0 does not support Memo field
216             if self.memo.as_ref().is_some_and(|m| !m.is_empty()) {
217                 return Err("version 0 MorphTx does not support Memo field");
218             }
219         }
220         MORPH_TX_VERSION_1 => {
221             // Version 1: FeeTokenID, Reference, Memo are all optional
222             // If FeeTokenID is 0, FeeLimit must not be set
223             if self.fee_token_id == 0 && self.fee_limit > U256::ZERO {
224                 return Err("version 1 MorphTx cannot have FeeLimit when FeeTokenID is 0");
225             }
226         }
227         _ => {
228             return Err("unsupported MorphTx version");
229         }
230     }
231     Ok(())
232 }
```

Listing 2.7: morph-reth/crates/primitives/src/transaction/morph_transaction.rs

Impact Valid `MorphTx V0` transactions accepted by Go nodes may be rejected by Rust nodes, causing inconsistent transaction admission.

Suggestion Update the validation logic to match the Go client implementation by allowing a zero-valued `reference` for `MorphTx V0`.

2.1.8 Lack of `gas_limit` check in function `maintain_morph_pool()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `maintain.rs`, the function `maintain_morph_pool()` revalidates pending `MorphTx` transactions on each canonical state change, checking token balances, fee parameters, and cumulative sender affordability. However, it does not verify whether a transaction's `gas_limit` exceeds the current block gas limit. If the block gas limit decreases after a transaction was initially admitted to the pool, that transaction may remain in the pool despite being unexecutable.

```
114 pub async fn maintain_morph_pool<Pool, Client>(pool: Pool, client: Client)
115 where
116     Pool: TransactionPool<Transaction = MorphPooledTransaction> + Clone,
117     Client: ChainSpecProvider<ChainSpec: MorphHardforks>
118         + StateProviderFactory
119         + CanonStateSubscriptions
120         + Clone
121         + 'static,
122 {
```

```rust
    let mut chain_events = client.canonical_state_stream();

    tracing::info!(target: "morph::txpool::maintain", "Starting MorphTx maintenance task");

    loop {
        // Wait for the next canonical state change
        let Some(event) = chain_events.next().await else {
            tracing::debug!(target: "morph::txpool::maintain", "Chain event stream ended");
            break;
        };

        let new_tip = event.tip();
        let block_number = new_tip.number();
        let block_timestamp = new_tip.timestamp();

        tracing::trace!(
            target: "morph::txpool::maintain",
            block_number,
            "Processing new block for MorphTx validation"
        );

        // Get the hardfork at this block
        let hardfork = client
            .chain_spec()
            .morph_hardfork_at(block_number, block_timestamp);

        // Collect all MorphTx transactions from the pool
        let all_txs = pool.all_transactions();
        let morph_txs: Vec<_> = all_txs
            .pending
            .iter()
            .chain(all_txs.queued.iter())
            .filter(|tx| tx.transaction.ty() == morph_primitives::MORPH_TX_TYPE_ID)
            .collect();

        if morph_txs.is_empty() {
            continue;
        }

        // Get state provider for the new tip
        let state_provider = match client.state_by_block_hash(new_tip.hash()) {
            Ok(provider) => provider,
            Err(err) => {
                tracing::warn!(
                    target: "morph::txpool::maintain",
                    %err,
                    "Failed to get state provider for MorphTx revalidation"
                );
                continue;
            }
        };

        let mut db = StateProviderDatabase::new(state_provider);
```

```
176
177         // Fetch L1 block info for fee calculation
178         let l1_block_info = match L1BlockInfo::try_fetch(&mut db, hardfork) {
179             Ok(info) => info,
180             Err(err) => {
181                 tracing::warn!(
182                     target: "morph::txpool::maintain",
183                     ?err,
184                     "Failed to fetch L1 block info for MorphTx revalidation"
185                 );
186                 continue;
187             }
188         };
189
190         tracing::trace!(
191             target: "morph::txpool::maintain",
192             count = morph_txs.len(),
193             "Revalidating MorphTx transactions"
194         );
195
196         // Group by sender and process in nonce order so affordability is validated cumulatively
              .
197         let mut txs_by_sender: HashMap<Address, Vec<_>> = HashMap::new();
198         for pooled_tx in morph_txs {
199             let sender = pooled_tx.transaction.sender();
200             txs_by_sender.entry(sender).or_default().push(pooled_tx);
201         }
202
203         // Revalidate each sender's MorphTx set and collect invalid ones
204         let mut to_remove: Vec<TxHash> = Vec::new();
205
206         for (sender, mut sender_txs) in txs_by_sender {
207             sender_txs.sort_by_key(|pooled_tx| pooled_tx.transaction.nonce());
208
209             // Initialize sender ETH budget once.
210             let eth_balance = match db.basic(sender) {
211                 Ok(Some(account)) => account.balance,
212                 Ok(None) => U256::ZERO,
213                 Err(err) => {
214                     tracing::warn!(
215                         target: "morph::txpool::maintain",
216                         ?sender,
217                         ?err,
218                         "Failed to get account balance"
219                     );
220                     continue;
221                 }
222             };
223
224             let mut budget = SenderBudget {
225                 eth_balance,
226                 token_balances: HashMap::new(),
227             };
```

```
228
229             for pooled_tx in sender_txs {
230                 let tx = &pooled_tx.transaction;
231                 // Access the consensus tx by reference (via Deref chain) instead of
232                 // cloning. Use the pool tx's cached EIP-2718 encoding for L1 fee.
233                 let consensus_tx = tx.transaction();
234
235                 let l1_data_fee = l1_block_info.calculate_tx_l1_cost(tx.encoded_2718(), hardfork)
                        ;
236
237                 // Use shared validation logic first with current sender ETH budget.
238                 let input = crate::MorphTxValidationInput {
239                     consensus_tx,
240                     sender,
241                     eth_balance: budget.eth_balance,
242                     l1_data_fee,
243                     base_fee_per_gas: new_tip.base_fee_per_gas(),
244                     hardfork,
245                 };
246
247                 let validation = match crate::validate_morph_tx(&mut db, &input) {
248                     Ok(v) => v,
249                     Err(err) => {
250                         tracing::debug!(
251                             target: "morph::txpool::maintain",
252                             tx_hash = ?tx.hash(),
253                             ?sender,
254                             ?err,
255                             "Removing MorphTx: validation failed"
256                         );
257                         to_remove.push(*tx.hash());
258                         break;
259                     }
260                 };
261
262                 let fields = consensus_tx.morph_fields();
263                 let state_token_balance = validation.token_info.as_ref().map(|info| info.balance)
                        ;
264                 let token_id = fields.as_ref().map(|f| f.fee_token_id);
265                 let fee_limit = fields.as_ref().map(|f| f.fee_limit);
266
267                 let affordable = if validation.uses_token_fee {
268                     consume_token_budget(
269                         &mut budget,
270                         consensus_tx.value(),
271                         token_id,
272                         fee_limit,
273                         validation.required_token_amount,
274                         state_token_balance,
275                     )
276                 } else {
277                     consume_eth_budget(
278                         &mut budget,
```

```
279                         consensus_tx.value(),
280                         consensus_tx.gas_limit(),
281                         consensus_tx.max_fee_per_gas(),
282                         l1_data_fee,
283                     )
284                 };
285                 if !affordable {
286                     tracing::debug!(
287                         target: "morph::txpool::maintain",
288                         tx_hash = ?tx.hash(),
289                         ?sender,
290                         uses_token_fee = validation.uses_token_fee,
291                         token_id = ?token_id,
292                         required_token_amount = ?validation.required_token_amount,
293                         "Removing MorphTx: insufficient cumulative sender budget"
294                     );
295                     to_remove.push(*tx.hash());
296                     break;
297                 }
298             }
299         }
300
301         // Remove invalid transactions and all higher-nonce descendants from the same sender.
302         // Using remove_transactions_and_descendants ensures that nonce-dependent txs are
                cleaned
303         // up immediately rather than becoming orphans that are re-validated every block.
304         if !to_remove.is_empty() {
305             let count = to_remove.len();
306             pool.remove_transactions_and_descendants(to_remove);
307             tracing::info!(
308                 target: "morph::txpool::maintain",
309                 count,
310                 block_number,
311                 "Removed invalid MorphTx transactions"
312             );
313         }
314     }
315 }
```

Listing 2.8: morph-reth/crates/txpool/src/maintain.rs

Impact Unexecutable transactions with `gas_limit` above the current block gas limit may persist in the mempool, consuming pool resources.

Suggestion Add a `gas_limit` check against the current block gas limit during `MorphTx` pool maintenance.

2.1.9 Non-Atomic canonical head lookup in function `current_head()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `builder.rs`, the function `current_head()` retrieves the canonical head in two sequential calls: function `chain_info()` returns the best block number and hash, then function `sealed_header_by_hash()` loads the corresponding header. If the canonical head advances between these two calls, the hash returned by function `chain_info()` may point to a header that is no longer accessible through the subsequent lookup, causing the function to fail with a "`canonical head not found`" error. In the Go client implemention, the equivalent operation reads the canonical head as a single object in one call.

```
884   fn current_head(&self) -> EngineApiResult<CanonicalHead>
885   where
886       Provider: HeaderProvider + BlockNumReader,
887   {
888       let info = self
889           .provider
890           .chain_info()
891           .map_err(|e| MorphEngineApiError::Database(e.to_string()))?;
892       let header = self
893           .provider
894           .sealed_header_by_hash(info.best_hash)
895           .map_err(|e| MorphEngineApiError::Database(e.to_string()))?
896           .ok_or_else(|| {
897               MorphEngineApiError::Internal(format!(
898                   "canonical head header {} ({}) not found",
899                   info.best_number, info.best_hash
900               ))
901           })?;
902
903       Ok(CanonicalHead {
904           number: info.best_number,
905           hash: info.best_hash,
906           timestamp: header.timestamp(),
907       })
908   }
```

Listing 2.9: morph-reth/crates/engine-api/src/builder.rs

Impact Concurrent head updates may cause transient failures in block assembly, validation, or import operations.

Suggestion Use an atomic or single-snapshot canonical head retrieval so the block hash and header are read from the same consistent state.

2.1.10 Underestimated token fee in `MorphTx` pool validation

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `morph_tx_validation.rs`, function `validate_morph_tx()` calculates the required ERC20 token fee using `effective_gas_price`, which is derived from the current

`base_fee_per_gas` and is always less than or equal to `max_fee_per_gas`. In the Go client implementation, the equivalent pool validation uses `GasPrice` (equal to `GasFeeCap`, i.e., `max_fee_per_gas`) for this calculation. If `base_fee_per_gas` increases between validation and execution, Rust client implementation may underestimate the required token amount during validation compared with the actual execution cost.

```
55  pub fn validate_morph_tx<DB: Database>(
56      db: &mut DB,
57      input: &MorphTxValidationInput<'_>,
58  ) -> Result<MorphTxValidationResult, MorphTxError> {
59      // Keep MorphTx structural validation in the shared path so both initial
60      // admission and background revalidation enforce the same invariants.
61      let morph_tx = match input.consensus_tx {
62          MorphTxEnvelope::Morph(signed) => signed.tx(),
63          _ => return Err(MorphTxError::InvalidTokenId),
64      };
65
66      if !input.hardfork.is_jade() && morph_tx.version == MORPH_TX_VERSION_1 {
67          return Err(MorphTxError::InvalidFormat {
68              reason: "MorphTx version 1 is not yet active (jade fork not reached)".to_string(),
69          });
70      }
71
72      if let Err(reason) = morph_tx.validate() {
73          return Err(MorphTxError::InvalidFormat {
74              reason: reason.to_string(),
75          });
76      }
77
78      let tx_value = morph_tx.value;
79      if tx_value > input.eth_balance {
80          return Err(MorphTxError::InsufficientEthForValue {
81              balance: input.eth_balance,
82              value: tx_value,
83          });
84      }
85
86      let fee_token_id = morph_tx.fee_token_id;
87      let fee_limit = morph_tx.fee_limit;
88
89      // Shared fee components used by both ETH-fee and token-fee branches.
90      let gas_limit = U256::from(morph_tx.gas_limit);
91      let max_fee_per_gas = U256::from(morph_tx.max_fee_per_gas);
92      let effective_gas_price = U256::from(morph_tx.effective_gas_price(input.base_fee_per_gas));
93      let gas_fee = gas_limit.saturating_mul(max_fee_per_gas);
94      let total_eth_fee = gas_fee.saturating_add(input.l1_data_fee);
95      let total_eth_cost = total_eth_fee.saturating_add(tx_value);
96
97      // fee_token_id == 0 means MorphTx uses ETH-fee path (reference/memo-only MorphTx).
98      if fee_token_id == 0 {
99          if total_eth_cost > input.eth_balance {
100             return Err(MorphTxError::InsufficientEthForValue {
```

```
                balance: input.eth_balance,
                value: total_eth_cost,
            });
        }
        return Ok(MorphTxValidationResult {
            uses_token_fee: false,
            token_info: None,
            required_token_amount: U256::ZERO,
            amount_to_pay: U256::ZERO,
        });
    }

    let token_info = TokenFeeInfo::load_for_caller(db, fee_token_id, input.sender, input.
        hardfork)
        .map_err(|err| MorphTxError::TokenInfoFetchFailed {
            token_id: fee_token_id,
            message: format!("{err:?}"),
        })?
        .ok_or(MorphTxError::TokenNotFound {
            token_id: fee_token_id,
        })?;

    // Check token is active
    if !token_info.is_active {
        return Err(MorphTxError::TokenNotActive {
            token_id: fee_token_id,
        });
    }

    // Check price ratio is valid
    if token_info.price_ratio.is_zero() {
        return Err(MorphTxError::InvalidPriceRatio {
            token_id: fee_token_id,
        });
    }

    let token_gas_fee = gas_limit.saturating_mul(effective_gas_price);
    let total_token_fee = token_gas_fee.saturating_add(input.l1_data_fee);
    let required_token_amount = token_info.eth_to_token_amount(total_token_fee);

    // Match REVM semantics:
    // - fee_limit == 0 => use token balance as effective limit
    // - fee_limit > balance => cap by token balance
    let effective_limit = if fee_limit.is_zero() || fee_limit > token_info.balance {
        token_info.balance
    } else {
        fee_limit
    };

    // Check token balance against effective limit.
    if effective_limit < required_token_amount {
        return Err(MorphTxError::InsufficientTokenBalance {
            token_id: fee_token_id,
```

```
153             token_address: token_info.token_address,
154             balance: effective_limit,
155             required: required_token_amount,
156         });
157     }
158
159     Ok(MorphTxValidationResult {
160         uses_token_fee: true,
161         token_info: Some(token_info),
162         required_token_amount,
163         amount_to_pay: required_token_amount,
164     })
165 }
```

Listing 2.10: morph-reth/crates/txpool/src/morph_tx_validation.rs

Impact A transaction may pass pool validation with an underestimated fee requirement but fail during execution when the `base_fee` increases.

Suggestion Use `max_fee_per_gas` instead of `effective_gas_price` for the upfront token fee calculation.

2.1.11 Lack of default `chain_id` handling in `MorphTx` RPC construction

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `transaction.rs`, function `try_build_morph_tx_from_request()` requires `chain_id` to be explicitly provided in the `RPC` request. If `chain_id` is absent, the function returns an error immediately. In the Go client implementation, RPC behavior defaults to the node's configured `chain_id` when it omits this field. The `MorphTx` construction path in Rust client implementation does not implement this fallback, causing valid requests without an explicit `chain_id` to be rejected.

```
230 fn try_build_morph_tx_from_request(
231     req: &alloy_rpc_types_eth::TransactionRequest,
232     fee_token_id: U64,
233     fee_limit: U256,
234     reference: Option<alloy_primitives::B256>,
235     memo: Option<alloy_primitives::Bytes>,
236 ) -> Result<Option<TxMorph>, &'static str> {
237     let fee_token_id_u16 = u16::try_from(fee_token_id.to::<u64>()).map_err(|_| "invalid token")
        ?;
238
239     // Check if this should be a MorphTx
240     let has_fee_token = fee_token_id_u16 > 0;
241     let has_reference = reference.is_some();
242     let has_memo = memo.as_ref().is_some_and(|m| !m.is_empty());
243
244     if !has_fee_token && !has_reference && !has_memo {
245         // No Morph-specific fields →standard Ethereum tx
```

```
246             return Ok(None);
247         }
248
249         // All MorphTx are constructed as Version 1
250         let version = morph_primitives::transaction::morph_transaction::MORPH_TX_VERSION_1;
251
252         // Now build the MorphTx
253         let chain_id = req
254             .chain_id
255             .ok_or("missing chain_id for morph transaction")?;
256         let gas_limit = req.gas.unwrap_or_default();
257         let nonce = req.nonce.unwrap_or_default();
258         let max_fee_per_gas = req.max_fee_per_gas.or(req.gas_price).unwrap_or_default();
259         let max_priority_fee_per_gas = req.max_priority_fee_per_gas.unwrap_or_default();
260         let access_list: AccessList = req.access_list.clone().unwrap_or_default();
261         let input = req.input.clone().into_input().unwrap_or_default();
262         let to = req.to.unwrap_or(TxKind::Create);
263
264         let morph_tx = TxMorph {
265             chain_id,
266             nonce,
267             gas_limit,
268             max_fee_per_gas,
269             max_priority_fee_per_gas,
270             to,
271             value: req.value.unwrap_or_default(),
272             access_list,
273             input,
274             fee_token_id: fee_token_id_u16,
275             fee_limit,
276             version,
277             reference,
278             memo,
279         };
280
281         // Validate all MorphTx constraints: version-specific rules, gas fee ordering,
282         // and memo length. This catches invalid combinations early at the RPC layer.
283         morph_tx.validate()?;
284
285         Ok(Some(morph_tx))
286     }
```

Listing 2.11: morph-reth/crates/rpc/src/eth/transaction.rs

Impact The valid local `MorphTx` requests can be rejected when `chain_id` is not explicitly specified.

Suggestion Fall back to the node's configured `chain_id` when `chain_id` is not specified in the RPC request.

2.1.12 Lack of RPC input validation in `MorphTx` construction

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `transaction.rs`, function `try_build_morph_tx_from_request()` does not perform two RPC `input` validations. First, when both the `data` and `input` fields are present with different values, the function silently selects `input` over `data` without reporting the conflict. In contrast, the Go client implementation rejects requests containing conflicting values for these fields. Second, when the `to` field is absent and the `input` is empty, the function constructs a contract creation transaction with no `initcode`. The Go client implementation rejects such requests instead of constructing the transaction. These validation differences may result in transaction construction behavior that differs from the Go client implementation.

```
230  fn try_build_morph_tx_from_request(
231      req: &alloy_rpc_types_eth::TransactionRequest,
232      fee_token_id: U64,
233      fee_limit: U256,
234      reference: Option<alloy_primitives::B256>,
235      memo: Option<alloy_primitives::Bytes>,
236  ) -> Result<Option<TxMorph>, &'static str> {
237      let fee_token_id_u16 = u16::try_from(fee_token_id.to::<u64>()).map_err(|_| "invalid token")
         ?;
238
239      // Check if this should be a MorphTx
240      let has_fee_token = fee_token_id_u16 > 0;
241      let has_reference = reference.is_some();
242      let has_memo = memo.as_ref().is_some_and(|m| !m.is_empty());
243
244      if !has_fee_token && !has_reference && !has_memo {
245          // No Morph-specific fields →standard Ethereum tx
246          return Ok(None);
247      }
248
249      // All MorphTx are constructed as Version 1
250      let version = morph_primitives::transaction::morph_transaction::MORPH_TX_VERSION_1;
251
252      // Now build the MorphTx
253      let chain_id = req
254          .chain_id
255          .ok_or("missing chain_id for morph transaction")?;
256      let gas_limit = req.gas.unwrap_or_default();
257      let nonce = req.nonce.unwrap_or_default();
258      let max_fee_per_gas = req.max_fee_per_gas.or(req.gas_price).unwrap_or_default();
259      let max_priority_fee_per_gas = req.max_priority_fee_per_gas.unwrap_or_default();
260      let access_list: AccessList = req.access_list.clone().unwrap_or_default();
261      let input = req.input.clone().into_input().unwrap_or_default();
262      let to = req.to.unwrap_or(TxKind::Create);
263
264      let morph_tx = TxMorph {
265          chain_id,
266          nonce,
267          gas_limit,
268          max_fee_per_gas,
```

```
269             max_priority_fee_per_gas,
270             to,
271             value: req.value.unwrap_or_default(),
272             access_list,
273             input,
274             fee_token_id: fee_token_id_u16,
275             fee_limit,
276             version,
277             reference,
278             memo,
279         };
280
281         // Validate all MorphTx constraints: version-specific rules, gas fee ordering,
282         // and memo length. This catches invalid combinations early at the RPC layer.
283         morph_tx.validate()?;
284
285         Ok(Some(morph_tx))
286     }
```

Listing 2.12: morph-reth/crates/rpc/src/eth/transaction.rs

Impact Malformed RPC requests with conflicting calldata fields or empty contract creation may be accepted by Rust client implementation but rejected by the Go client implementation, potentially causing inconsistent transaction construction behavior.

Suggestion Reject requests where `data` and `input` fields are both present with conflicting values, and reject contract creation requests with empty calldata.

2.1.13 Inconsistent gas estimation behavior between Rust and Go clients implementation

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In file `transaction.rs`, function `try_build_morph_tx_from_request()` determines whether a request should enter the token-fee path by checking whether `fee_token_id` is greater than 0. As a result, when a request specifies a `fee_token_id` value of zero, the Rust client implementation treats the request as a native-fee transaction and does not construct a token-fee transaction.

In contrast, the Go client implementation interprets the presence of `fee_token_id` as an indication that token-fee payment is intended. Consequently, requests with a `fee_token_id` value of zero are rejected during gas estimation.

As a result, gas estimation requests specifying a `fee_token_id` value of zero are handled differently by the Rust and Go client implementations.

```
230     fn try_build_morph_tx_from_request(
231         req: &alloy_rpc_types_eth::TransactionRequest,
232         fee_token_id: U64,
233         fee_limit: U256,
```

```
234         reference: Option<alloy_primitives::B256>,
235         memo: Option<alloy_primitives::Bytes>,
236     ) -> Result<Option<TxMorph>, &'static str> {
237         let fee_token_id_u16 = u16::try_from(fee_token_id.to::<u64>()).map_err(|_| "invalid token")
                ?;
238 
239         // Check if this should be a MorphTx
240         let has_fee_token = fee_token_id_u16 > 0;
241         let has_reference = reference.is_some();
242         let has_memo = memo.as_ref().is_some_and(|m| !m.is_empty());
243 
244         if !has_fee_token && !has_reference && !has_memo {
245             // No Morph-specific fields →standard Ethereum tx
246             return Ok(None);
247         }
248 
249         // All MorphTx are constructed as Version 1
250         let version = morph_primitives::transaction::morph_transaction::MORPH_TX_VERSION_1;
251 
252         // Now build the MorphTx
253         let chain_id = req
254             .chain_id
255             .ok_or("missing chain_id for morph transaction")?;
256         let gas_limit = req.gas.unwrap_or_default();
257         let nonce = req.nonce.unwrap_or_default();
258         let max_fee_per_gas = req.max_fee_per_gas.or(req.gas_price).unwrap_or_default();
259         let max_priority_fee_per_gas = req.max_priority_fee_per_gas.unwrap_or_default();
260         let access_list: AccessList = req.access_list.clone().unwrap_or_default();
261         let input = req.input.clone().into_input().unwrap_or_default();
262         let to = req.to.unwrap_or(TxKind::Create);
263 
264         let morph_tx = TxMorph {
265             chain_id,
266             nonce,
267             gas_limit,
268             max_fee_per_gas,
269             max_priority_fee_per_gas,
270             to,
271             value: req.value.unwrap_or_default(),
272             access_list,
273             input,
274             fee_token_id: fee_token_id_u16,
275             fee_limit,
276             version,
277             reference,
278             memo,
279         };
280 
281         // Validate all MorphTx constraints: version-specific rules, gas fee ordering,
282         // and memo length. This catches invalid combinations early at the RPC layer.
283         morph_tx.validate()?;
284 
285         Ok(Some(morph_tx))
```

```
286    }
```

Listing 2.13: morph-reth/crates/rpc/src/eth/transaction.rs

Impact Identical gas estimation requests may be accepted by the Rust client implementation but rejected by the Go client implementation when the value of `fee_token_id` is zero, resulting in inconsistent behavior across client implementations.

Suggestion Ensure consistent handling of requests specifying a `fee_token_id` value of zero across Rust and Go client implementations.

Feedback from the project The project clarifies that the issue will be addressed in the Go client, as the current behavior of the Rust client is considered the intended implementation.

2.2 Recommendation

2.2.1 Update the precompile set comment reference

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `precompiles.rs`, the comment for the function `bernoulli()` links to `PrecompiledContractsMorph203` in the Go client implementation, while the function itself implements the `Bernoulli` precompile set. The correct reference should be `PrecompiledContractsBernoulli`.

These two precompile sets are distinct and have different configurations for `ripemd160 (0x03)` and `blake2f (0x09)`. The incorrect reference may mislead future reviewers when comparing the in Rust and Go client implementation precompile definitions.

```
248    /// Returns precompiles for Bernoulli hardfork.
249    ///
250    /// Based on Berlin with ripemd160 (0x03) and blake2f (0x09) replaced by disabled stubs.
251    /// All 9 Berlin addresses are present (so they get warmed via EIP-2929), but 0x03/0x09
252    /// consume all forwarded gas and return failure when called.
253    ///
254    /// Matches: <https://github.com/morph-l2/go-ethereum/blob/main/core/vm/contracts.go#L136-L148>
255    pub fn bernoulli() -> &'static Precompiles {
256        static INSTANCE: OnceLock<Precompiles> = OnceLock::new();
257        INSTANCE.get_or_init(|| {
258            // Start from Berlin (9 precompiles including 0x03 and 0x09).
259            let mut precompiles = Precompiles::berlin().clone();
260
261            // Replace ripemd160 (0x03) and blake2f (0x09) with disabled stubs.
262            // This keeps them in warm_addresses() so EIP-2929 warms them (100 gas instead of
263            // 2600 cold), matching go-ethereum's PrecompiledContractsBernoulli behavior.
264            precompiles.extend([
265                Precompile::new(
266                    PrecompileId::Ripemd160,
267                    addresses::RIPEMD160,
268                    ripemd160_disabled,
269                ),
270                Precompile::new(PrecompileId::Blake2F, addresses::BLAKE2F, blake2f_disabled),
```

```
271         ]);
272
273         // Replace modexp (0x05) with 32-byte input limit wrapper.
274         // go-ethereum's Bernoulli modexp has eip2565=true but neither eip7823 nor eip7883,
275         // which enforces base/exp/mod <= 32 bytes. Berlin modexp in revm has no such limit.
276         precompiles.extend([Precompile::new(
277             PrecompileId::ModExp,
278             addresses::MODEXP,
279             modexp_with_32byte_limit,
280         )]);
281
282         precompiles
283     })
284 }
```

Listing 2.14: morph-reth/crates/revm/src/precompiles.rs

Suggestion Update the comment to reference the correct precompile set in the Go client implementation, and ensure that the linked line numbers point to the corresponding implementation.

2.2.2 Apply `MorphTx` validation in the RPC simulation path

Status Confirmed

Introduced by `Version 1`

Description In file `transaction.rs`, the function `try_into_tx_env()` converts an RPC request into a `MorphTxEnv` for functions `eth_call()` and `eth_estimateGas()`. Unlike the real transaction construction path (in function `try_build_morph_tx_from_request()`), it does not invoke function `TxMorph::validate()` or perform an equivalent structural check. Although this path only serves simulation and does not submit transactions on-chain, function `eth_call()` and function `eth_estimateGas()` should enforce the same `MorphTx` structural rules to prevent simulations of transactions that would be rejected during actual construction or submission.

```
162 fn try_into_tx_env(
163     self,
164     evm_env: &EvmEnv<Spec, MorphBlockEnv>,
165 ) -> Result<MorphTxEnv, Self::Err> {
166     let fee_token_id = self.fee_token_id;
167     let fee_limit = self.fee_limit;
168     let reference = self.reference;
169     let memo = self.memo;
170     let inner = self.inner;
171
172     let inner_tx_env = inner.try_into_tx_env(evm_env).map_err(EthApiError::from)?;
173
174     let mut tx_env = MorphTxEnv::new(inner_tx_env);
175     tx_env.fee_token_id = match fee_token_id {
176         Some(id) => Some(
177             u16::try_from(id.to::<u64>())
178                 .map_err(|_| EthApiError::InvalidParams("invalid token".to_string()))?,
```

```
179             ),
180             None => None,
181         };
182         tx_env.fee_limit = fee_limit;
183         tx_env.reference = reference;
184         tx_env.memo = memo.clone();
185
186         // Determine if this is a MorphTx based on Morph-specific fields
187         let is_morph_tx = fee_token_id.is_some_and(|id| id.to::<u64>() > 0)
188             || reference.is_some()
189             || memo.as_ref().is_some_and(|m| !m.is_empty());
190
191         if is_morph_tx {
192             tx_env.inner.tx_type = morph_primitives::MORPH_TX_TYPE_ID;
193             tx_env.version =
194                 Some(morph_primitives::transaction::morph_transaction::MORPH_TX_VERSION_1);
195         }
196
197         // Required by `MorphEthApi::caller_gas_allowance` (eth/call.rs) to
198         // recover the L1 data fee when capping `eth_estimateGas` allowance.
199         tx_env.rlp_bytes = Some(tx_env.encode_for_l1_fee(evm_env.cfg_env.chain_id));
200
201         Ok(tx_env)
202     }
```

Listing 2.15: morph-reth/crates/rpc/src/eth/transaction.rs

Suggestion Reuse the `MorphTx` structural validation in function `try_into_tx_env()` to match the real transaction construction path.

Feedback from the project The project clarifies that this code path is used exclusively by the `eth_call()` and `eth_estimateGas()` RPC methods. RPC request conversion already handles `MorphTx` field parsing, transaction version determination, and L1 fee encoding. As a result, invoking function `TxMorph::validate()` in this path would duplicate validation logic with limited practical benefit.

2.2.3 Handle database read errors in storage `original_value` restoration

Status Confirmed

Introduced by `Version 1`

Description In file `evm.rs`, functions `sload_morph()` and `sstore_morph()` restore the `original_value` of storage slots that were modified during token fee deduction. The restoration reads the committed value from the database and only applies the correction when the read succeeds. If the database read fails, the error is silently ignored and execution continues with a potentially incorrect `original_value`. Since the correctness of `EIP-2200` gas accounting and refund calculation depends on `original_value`, a failed restoration could cause gas charges to diverge from the expected behavior.

```
96   fn sload_morph<DB: Database>(context: InstructionContext<'_, MorphContext<DB>, EthInterpreter>)
         {
```

```
97      let Some(([], index)) = StackTr::popn_top::<0>(&mut context.interpreter.stack) else {
98          context.interpreter.halt_underflow();
99          return;
100     };
101
102     let target = context.interpreter.input.target_address;
103     let key = *index;
104
105     let additional_cold_cost = context.host.gas_params().cold_storage_additional_cost();
106     let skip_cold = context.interpreter.gas.remaining() < additional_cold_cost;
107     let res = context.host.sload_skip_cold_load(target, key, skip_cold);
108
109     match res {
110         Ok(storage) => {
111             if storage.is_cold {
112                 // Read the true committed value from DB (hits State<DB> cache, O(1)).
113                 // This matches go-eth's GetCommittedState() returning the un-modified DB value.
114                 let db_original = context.host.journaled_state.database.storage(target, key);
115                 if let Ok(db_original) = db_original
116                     && let Some(acc) = context.host.journaled_state.inner.state.get_mut(&target)
117                     && let Some(slot) = acc.storage.get_mut(&key)
118                     && slot.original_value != db_original
119                 {
120                     slot.original_value = db_original;
121                 }
122
123                 if !context
124                     .interpreter
125                     .gas
126                     .record_regular_cost(additional_cold_cost)
127                 {
128                     context.interpreter.halt_oog();
129                     return;
130                 }
131             }
132
133             *index = storage.data;
134         }
135         Err(LoadError::ColdLoadSkipped) => context.interpreter.halt_oog(),
136         Err(LoadError::DBError) => context.interpreter.halt_fatal(),
137     }
138 }
```

Listing 2.16: morph-reth/crates/revm/src/evm.rs

```
96  fn sload_morph<DB: Database>(context: InstructionContext<'_, MorphContext<DB>, EthInterpreter>)
        {
97      let Some(([], index)) = StackTr::popn_top::<0>(&mut context.interpreter.stack) else {
98          context.interpreter.halt_underflow();
99          return;
100     };
101
102     let target = context.interpreter.input.target_address;
```

```
103         let key = *index;
104
105         let additional_cold_cost = context.host.gas_params().cold_storage_additional_cost();
106         let skip_cold = context.interpreter.gas.remaining() < additional_cold_cost;
107         let res = context.host.sload_skip_cold_load(target, key, skip_cold);
108
109         match res {
110             Ok(storage) => {
111                 if storage.is_cold {
112                     // Read the true committed value from DB (hits State<DB> cache, O(1)).
113                     // This matches go-eth's GetCommittedState() returning the un-modified DB value.
114                     let db_original = context.host.journaled_state.database.storage(target, key);
115                     if let Ok(db_original) = db_original
116                         && let Some(acc) = context.host.journaled_state.inner.state.get_mut(&target)
117                         && let Some(slot) = acc.storage.get_mut(&key)
118                         && slot.original_value != db_original
119                     {
120                         slot.original_value = db_original;
121                     }
122
123                     if !context
124                         .interpreter
125                         .gas
126                         .record_regular_cost(additional_cold_cost)
127                     {
128                         context.interpreter.halt_oog();
129                         return;
130                     }
131                 }
132
133                 *index = storage.data;
134             }
135             Err(LoadError::ColdLoadSkipped) => context.interpreter.halt_oog(),
136             Err(LoadError::DBError) => context.interpreter.halt_fatal(),
137         }
138     }
139
140     /// Morph custom SSTORE opcode.
141     ///
142     /// Twin of [`sload_morph`]: revm's standard SSTORE warms a cold slot through
143     /// the same `mark_warm_with_transaction_id()` path as SLOAD, so forced-cold
144     /// token-fee slots need the same `original_value` restoration before
145     /// `sstore_dynamic_gas()` reads it for EIP-2200 accounting.
146     ///
147     /// Without this, a main tx that writes a fee-deducted slot WITHOUT first
148     /// SLOADing it sees a "clean" slot (2900 gas SSTORE_RESET, no refund)
149     /// instead of a "dirty" slot (100 gas SLOAD_GAS plus refund), causing the
150     /// same 2800-gas-per-write divergence vs go-eth that `sload_morph` fixes.
151     ///
152     /// Uses DB-direct lookup (no per-tx runtime map needed).
153     fn sstore_morph<DB: Database>(context: InstructionContext<'_, MorphContext<DB>, EthInterpreter
        >) {
154         if context.interpreter.runtime_flag.is_static() {
```

```
155             context
156                 .interpreter
157                 .halt(InstructionResult::StateChangeDuringStaticCall);
158             return;
159         }
160
161         let Some([index, value]) = StackTr::popn::<2>(&mut context.interpreter.stack) else {
162             context.interpreter.halt_underflow();
163             return;
164         };
165
166         let target = context.interpreter.input.target_address;
167         let spec_id = context.interpreter.runtime_flag.spec_id();
168
169         if spec_id.is_enabled_in(ISTANBUL)
170             && context.interpreter.gas.remaining() <= context.host.gas_params().call_stipend()
171         {
172             context
173                 .interpreter
174                 .halt(InstructionResult::ReentrancySentryOOG);
175             return;
176         }
177
178         if !context
179             .interpreter
180             .gas
181             .record_regular_cost(context.host.gas_params().sstore_static_gas())
182         {
183             context.interpreter.halt_oog();
184             return;
185         }
186
187         let mut state_load = if spec_id.is_enabled_in(BERLIN) {
188             let additional_cold_cost = context.host.gas_params().cold_storage_additional_cost();
189             let skip_cold = context.interpreter.gas.remaining() < additional_cold_cost;
190             match context
191                 .host
192                 .sstore_skip_cold_load(target, index, value, skip_cold)
193             {
194                 Ok(load) => load,
195                 Err(LoadError::ColdLoadSkipped) => {
196                     context.interpreter.halt_oog();
197                     return;
198                 }
199                 Err(LoadError::DBError) => {
200                     context.interpreter.halt_fatal();
201                     return;
202                 }
203             }
204         } else {
205             let Some(load) = context.host.sstore(target, index, value) else {
206                 context.interpreter.halt_fatal();
207                 return;
```

```
208             };
209             load
210         };
211
212         // Morph fix: on cold access, restore original_value from the DB-committed value.
213         // Mirrors sload_morph. Only fires on cold path; zero overhead on warm SSTOREs.
214         if state_load.is_cold {
215             let db_original = context.host.journaled_state.database.storage(target, index);
216             if let Ok(db_original) = db_original
217                 && state_load.data.original_value != db_original
218             {
219                 state_load.data.original_value = db_original;
220                 if let Some(acc) = context.host.journaled_state.inner.state.get_mut(&target)
221                     && let Some(slot) = acc.storage.get_mut(&index)
222                 {
223                     slot.original_value = db_original;
224                 }
225             }
226         }
227
228         let is_istanbul = spec_id.is_enabled_in(ISTANBUL);
229         let dynamic_gas = context.host.gas_params().sstore_dynamic_gas(
230             is_istanbul,
231             &state_load.data,
232             state_load.is_cold,
233         );
234         if !context.interpreter.gas.record_regular_cost(dynamic_gas) {
235             context.interpreter.halt_oog();
236             return;
237         }
238
239         context.interpreter.gas.record_refund(
240             context
241                 .host
242                 .gas_params()
243                 .sstore_refund(is_istanbul, &state_load.data),
244         );
245     }
```

Listing 2.17: morph-reth/crates/revm/src/evm.rs

Suggestion Propagate the database read error and halt execution when the committed value cannot be retrieved.

Feedback from the project The project clarifies that the committed storage read is intended only as a best-effort correction of `original_value` following token-fee related storage updates. Failures in the primary `sload` and `sstore` database operations already halt execution, whereas failures in this auxiliary correction path are intentionally treated as non-fatal. Treating such failures as fatal would introduce additional complexity without a corresponding benefit.

2.3 Note

2.3.1 Trusted token and oracle assumptions

Introduced by Version 1

Description Protocol supported gas tokens are assumed to be whitelisted, trusted, and standards-compliant ERC20 tokens. Under this assumption, the function `balanceOf()` function returns a valid 32-byte balance, the `transfer()` function follows normal ERC20 semantics, and token behavior does not depend on unexpected caller-specific logic. The chain oracle is also assumed to provide correct and timely pricing data for token fee conversion.

Under these assumptions, the observed implementation differences in balance queries are unlikely to affect supported gas tokens. Specifically, in function `query_balance_via_system_call()`, the Rust client implementation falls back to function `system_call_one()` when the cached balance slot is unavailable, while the Go client implementation uses function `StaticCall()` with the user address as the caller. In addition, the Rust client implementation treats short or failed `balanceOf()` results as zero, whereas the Go client implementation returns an error when the returned data is shorter than expected.

A similar caller difference also exists in function `evm_call_balance_of()`. The Rust client implementation constructs the correct calldata, but executes the internal call with `Address::ZERO` as the caller instead of the queried account.

For standard whitelisted gas tokens, these differences should not affect the returned balance.

2.3.2 Trusted upstream `Revm` baseline

Introduced by Version 1

Description The audit focused on the `Morph` specific changes and the files explicitly included in the audit scope, rather than the entire codebase. As an assumption, the upstream Rust execution client logic based on `paradigmxyz/reth v2.2.0`[1] was treated as a trusted baseline dependency. This baseline covers the `Reth` execution stack, together with the `Revm` derived EVM logic used by `Reth`. These upstream components were assumed to correctly implement the expected Ethereum execution semantics.

[1] https://github.com/paradigmxyz/reth/tree/88505c7fcbfdebfd3b56d88c86b62e950043c6c4


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS