OpenZeppelin | security

EVM Equivalence Audit

January 15, 2025

Table of Contents

Summary

Type	Layer 2
Timeline	From 2024-10-28 To 2024-12-13
Languages	Yul, Solidity

Total Issues	39 (33 resolved, 3 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	5 (4 resolved, 1 partially resolved)
Medium Severity Issues	10 (9 resolved)
Low Severity Issues	10 (8 resolved, 1 partially resolved)
Notes & Additional Information	14 (12 resolved, 1 partially resolved)

Scope

We audited the matterlabs/era-contracts repository at commit 397092d.

In scope were the following files:

```
era-contracts/system-contracts/
├── contracts
│   ├── EvmEmulator.yul
│   └── EvmGasManager.yul
└── evm-emulator
    ├── EvmEmulator.template.yul
    ├── EvmEmulatorFunctions.template.yul
    ├── EvmEmulatorLoop.template.yul
    └── EvmEmulatorLoopUnusedOpcodes.template.yul
```

We also diff-audited the matterlabs/era-contracts repository between base commit 658713f and head commit ed6f4d1.

System Overview

To facilitate developers relying on EVM bytecode support, ZKsync Era has introduced an EVM execution mode via emulation on top of EraVm.

Core System: The primary system remains EraVM, allowing for the deployment and execution of native EraVM contracts. The unit of gas for all executions continues to be the native EraVM gas (ergs).

EVM Contract Execution: The platform now also supports the deployment and execution of EVM contracts. EraVm and EVM contracts are distinguished by a special marker in their bytecode hash. When EVM bytecode is invoked, the virtual machine invokes the predefined `EvmEmulator` contract. This emulator loads, interprets, and executes the EVM bytecode in a manner consistent with Ethereum's execution environment, with adjustments made for the peculiarities of EraVM. This is achieved by emulating the EVM opcodes. Specifically, the implementation adheres to the EVM specification after the Cancun upgrade fork.

Environment Agnostic: The EVM environment operates independently of EraVM, meaning that EraVM contracts are unaware of the EVM emulation. When an EraVM contract calls an EVM contract, the calldata is passed as it is, without any modifications. Similarly, when an EVM contract returns data to an EraVM contract, the returndata is returned as it is, without any further changes. Any modification needed is handled solely by the system contracts and the emulator. A special system contract, `EvmGasManager`, behaves like a helper contract for the emulator and is responsible for handling the hot/cold storage slots and accounts. It also handles the `EvmFrame`s in cooperation with the emulator, which are needed in order to pass the `isStatic` flag and remaining EVM gas information between the EVM call frames.

Interaction Between EraVM and EVM:

The following is the basic schema for the interaction between EraVm and EVM contracts:

- A transaction always starts with the execution of `Bootloader`, which initiates a subsequent call.
- When an EraVM contract calls an EVM contract, the actual target contract's bytecode is replaced with the bytecode of the EVM emulator, which loads the target contract's bytecode and initiates the interpretation process.

- When an EVM contract calls an EraVM contract, it is treated as a standard call to a native EraVM contract.
- The gas units from the EVM environment are converted to ergs using a fixed conversion ratio and vice versa.

This approach ensures seamless interoperability between native EraVM and EVM contracts while preserving the distinct execution contexts of each.

Notable Differences From the Ethereum Execution Environment

Gas Costs

When an emulated EVM contract is executed, the gas charged is the native gas of ZKsync Era, called ergs. In order to remain as close as possible to the EVM behavior, the emulator keeps track of the EVM gas following the EVM opcodes' specification. When an EVM contract is triggered for the first time by an EraVM contract, the ergs passed are converted to EVM gas units using a specified ratio. As the bytecode execution moves forward, the emulator subtracts the corresponding gas costs from the total gas left. If another EVM contract is called during the execution, the EVM gas passed to the callee frame respects the EVM gas left to the current execution frame and other EVM rules (e.g., the 63/64th rule).

If an EraVM contract is called during the execution, the ergs passed for the call also respect the EVM gas left to the current execution (converted appropriately), even if EVM gas costs are generally expected to differ from the actual charged ergs. This design is needed to ensure that any potential subsequent call to an EVM contract receives a lower EVM gas amount than the EVM gas left at the origin EVM contract. Note that, per this design, developers should be extra cautious regarding the gas provided to the nested calls originating from their contracts. This is to avoid out-of-gas errors or gas griefing attacks within the EraVM environment.

EVM <-> EraVM `delegatecall` Is Not Allowed

While calls between EVM and EraVM contracts are supported, `delegatecall`s are not. This is important to ensure the integrity of the emulator's processes and bookkeeping that follows the EVM specification closely (e.g., the cold/warm storage slots). Developers need to be mindful of this design, especially when it comes to certain design patterns that involve proxy contracts.

Subset of EVM Precompiles Is Supported

Currently, 6 out of the 10 EVM precompiles are supported. Specifically, the ones not supported are: `RIPEMD-160`, `modexp`, `blake2f`, and KZG point evaluation.

Security Model and Trust Assumptions

The EVM emulator contract can be upgraded by privileged and trusted entities who administer the system. This is important for providing code fixes for potential security issues. Other than that, the EVM emulator and the diffs audited do not introduce any new security assumptions to the system.

High Severity

H-01 Wrong `CALL` Implementation Under Static Context

According to the EVM opcodes specification, the `CALL` opcode should revert if the execution context is static and the `value` parameter is not zero. In the `EvmEmulatorLoop` template, the `CALL` opcode implementation checks whether the current execution context is static and performs a normal call or a static call by calling the `performCall` and `performStaticCall` functions accordingly.

In case of a static context, `value` is never checked against zero. Specifically, `performStaticCall` extracts only 6 items out of the stack, essentially as many as expected from a `STATICCALL`. However, even in a static context, the `CALL` opcode requires 7 input items on the stack. Consequently, the input parameters are wrongly retrieved from the stack in this scenario, essentially loading `value` to `argsOffset`, `argsOffset` to `argsSize`, and so on.

Consider popping the input arguments out of the `performCall` and `performStaticCall` functions so that they are properly handled according to each opcode's specification. In addition, in order to remain compatible with the EVM specification, consider always checking the `value` parameter when executing the `CALL` opcode in a static context and reverting in case of a non-zero value.

Update: *Resolved in pull request #1091 at commit 6fe1257.*

H-02 Possible Arbitrary Code Execution in EVM Contracts

A smart contract on the EVM runs a sequence of instructions known as opcodes. Execution generally proceeds one instruction at a time, moving forward through the code. To support loops and conditional branching, the EVM uses `JUMP` and `JUMPI` instructions, which alter the instruction pointer to a specified position in the contract's bytecode. Under normal conditions, the jump targets must be valid positions within the deployed contract bytecode.

In the `EvmEmulator` contract, the `JUMP` and `JUMPI` instructions lack sufficient validation. While there is a check to ensure that the pointer does not exceed the code length, there is no corresponding check to ensure that it does not point to a location before the `BYTECODE_OFFSET`. Since the code and data structures, including the stack, are stored in the same EraVM memory space, a carefully chosen `counter` can wrap around due to a lack of overflow checks. This can redirect execution into memory regions before the bytecode offset, including the emulated stack that is possible to control and malform. As a result, it may be possible to execute arbitrary instructions, enabling unpredictable execution flows.

Consider implementing a strict validation check which ensures that the instruction pointer always targets a position within the contract's bytecode.

Update: *Resolved in pull request #1159 at commit 0419d3e.*

H-03 Discrepancy in `getCodeHash` Compared to EVM `extcodehash` Behavior

The `AccountCodeStorage` contract is responsible for storing the code hashes of accounts. There are two functions for retrieving an account's code hash: `getRawCodeHash`, which simply reads the value stored under the account's slot, and `getCodeHash`, which simulates the behavior of the EVM `extcodehash` opcode. However, the behavior of the `getCodeHash` function differs from that of the EVM `extcodehash` opcode when the account has no code, a zero nonce, and a **non-zero** balance. In the EVM, `extcodehash` should return the hash of an empty string in this scenario. However, in the codebase, `0x00` is returned.

Likewise, the EVM emulator includes a specific check for a zero address and returns zero in that case. However, this approach can be error-prone because if the zero address has a zero balance, the expected result should be zero, and if the zero address has a non-zero balance, the result should be the hash of an empty string. It is also worth noting that during the construction of an EVM contract, the construction bytecode (`0x020100...00`) can be retrieved for the account. This behavior deviates from the EVM's expected `extcodehash` behavior.

Consider standardizing the behavior of the codebase to align it with that of the EVM and have it accurately mimic the account code hash functionality.

Update: *Partially resolved in pull request #1125 at commit c4c964b. The Matter Labs team stated:*

> *Fixed by reimplementing EXTCODEHASH inside the emulator.*

H-04 Complete EVM Environment Abort for `CALL` with Insufficient Value

The `CALL` opcode in an EVM environment invokes a target account with specified parameters, including data, gas, and value. According to its specification, if the transferred value exceeds the caller's available balance, the call should fail but not revert the overall execution context. Instead, it should return zero, produce empty returndata, and refund the appropriate gas stipend. ZKsync's compiler adds conditional behavior based on the call's value argument: if the value is zero, the call is directly made to the target contract. However, if the value is non-zero, the call routes through the `MsgValueSimulator`.

In the current EVM emulator implementation, when the caller's balance is insufficient, the entire execution chain is inadvertently aborted. This occurs because the caller's balance is not verified before initiating the EraVM's `CALL` opcode. Since the value is not zero, the call goes through `MsgValueSimulator`, triggering the fallback function of `MsgValueSimulator` which subsequently calls `transferFromTo` in `L2BaseToken`. If the caller lacks the required balance, the call to `L2BaseToken` fails and `MsgValueSimulator` reverts with `revert(0, 0)`. As a result, `_saveReturndataAfterEVMCall` processes empty returndata, leading to `abortEvmEnvironment` and halting the entire call chain instead of simply returning zero and empty returndata.

Consider performing a balance check of the caller before executing the `CALL` opcode. This ensures that if the requested value cannot be covered, the call correctly fails without reverting the entire context, preserving the intended EVM behavior of returning zero success status and empty returndata.

Update: *Resolved in pull request #1108 at commit c74c6ec.*

H-05 Incorrect Bytecode Assumption for EraVM Contract in `fetchDeployedCode`

Two types of code hashes are present in the protocol: type 1 for EraVM contract code hashes and type 2 for EVM contract code hashes. During EVM contract deployment, the code hash is derived from the padded bytecode containing the length prefix, the bytecode itself, and any necessary padding to align with blockchain requirements. System flags, the code hash version, and the code length are stored in the most significant bits.

In the EVM emulator, when the `extcodecopy` opcode executes, it calls the `fetchDeployedCode` function, which then invokes `CodeOracle` to retrieve the contract

bytecode. Here, it is assumed that the first word represents the account length, but this only applies to EVM-type contracts as EraVM contracts do not have this length prefix. This causes the emulator to treat the first 32 bytes of the hash as a length value and can lead to bytecode misalignment, incorrect behavior, and potential overflow when calculating offsets. Large values can trigger out-of-bounds attempts at `returndatacopy`, while other values can lead to skipping parts of the bytecode.

Consider not assuming that the first word returned by `CodeOracle` represents the contract length. Instead, rely on the length information derived from the raw code hash. Take into account the fact that EraVM contracts store their length in words, whereas EVM contracts store it in bytes.

Update: *Resolved in pull request #1157 at commit c396c03 and in pull request #1196 at commit 875d0a5. The Matter Labs team stated:*

> *Prefixes are removed, length is encoded in the hash.*

Medium Severity

M-01 Missing Stack Overflow Check In `dupStackItem`

In the `EvmEmulatorFunctions` template, the `dupStackItem` function duplicates a specific stack item by copying it and pushing it into the stack. However, it is not checked that the stack will not overflow before duplicating the item. As a consequence, it is possible to overflow beyond the dedicated stack's space.

Consider checking that there is available stack space before inserting another item via the `dupStackItem` function.

Update: *Resolved in pull request #1087 at commit ba50e19.*

M-02 Incorrect Implementation of the `checkMemIsAccessible` Function

The implemented EVM emulator sets a bounded memory space, considering the maximum possible memory expansion due to each block's gas limit. The `checkMemIsAccessible`

function is responsible for checking that a requested part of the memory, as defined by a memory offset and the data size, is within these bounds. However, when performing the check within `checkMemIsAccessible`, the offset is mistakenly not considered in relation to the memory's starting slot. Thus, the check always underestimates the maximum memory slot requested.

Consider adding `MEM_OFFSET` to the offset argument of the `checkMemIsAccessible` function in order to properly check for potential out-of-bounds memory access.

Update: *Resolved in pull request #1082 at commit 8e8323d. The Matter Labs team stated:*

> *This issue has no real impact, as this check is intended to simplify out-of-gas errors when trying to expand memory to too large values. Given the actual block gas limit, the current implementation also works. However, the implementation has been fixed.*

M-03 Callee Frame Charged With Address Decommitment Cost

The EVM emulator adheres closely to the EVM gas cost specification to accurately simulate the EVM environment. As a result, the gas costs during an emulated transaction are expected to be overestimated. Even if the user is not actually charged the EVM gas costs, these costs cap the gas passed to a nested call. When a call to a zkEVM contract is executed, there is an additional gas cost for the decommitment of the called address. Currently, the decommitment gas cost further limits the gas passed to the callee frame, even if it is not an emulated EVM cost. As a result, the gas passed to the callee frame could be significantly underestimated, potentially leading to out-of-gas errors.

In order to prevent unreasonable out-of-gas errors, consider excluding the decommitment gas cost when calculating the gas passed to the zkEVM callee frame. Alternatively, consider giving an extra gas stipend for this cost if it is expected to be charged within the callee frame.

Update: *Resolved in pull request #1086 at commit b649f5f, pull request #1196 at commit 3c77d4a. The Matter Labs team stated:*

> *Explicit address decommitment cost has been removed. Additionally, extra stipend was added, to cover call to Empty and DefaultAccount contracts.*

M-04 Possible to Zero Out Stack, Bytecode, or Memory in an EVM Contract

The EVM executes smart contracts by manipulating data in memory, stack, and storage. The `extcodecopy` operation is typically used to load another contract's bytecode into memory. This process involves specifying an offset and a length, after which the code is copied and any remaining space is filled with zeroes.

In the `extcodecopy` implementation of the `EvmEmulator` contract, if the requested length exceeds the actual code length of the target contract, the `$llvm_AlwaysInline_llvm$_memsetToZero` function is called, which sets the remaining memory to zero. Currently, the starting point of the zeroing process does not include the `MEM_OFFSET` and only accounts for the `dstOffset` and `copiedLen`. As a result, it is possible to target specific memory regions—such as system variables, stack values (if `dstOffset` equals `STACK_OFFSET`), contract memory areas, contract bytecode, or previously returned data length—and overwrite them with zeroes under certain conditions.

Consider including the `MEM_OFFSET` when calculating the starting point for zeroing the memory region. This ensures that system variables and other memory areas are not unintentionally overwritten.

Update: *Resolved in pull request #1115 at commit 37874c7.*

M-05 `extcodecopy` Opcode Behavior Diverges From EVM

The `extcodecopy` opcode in the EVM is used to copy code from a given address to the memory of the current context. An address in the EVM is represented as 20 bytes (160 bits). In the current EVM Emulator implementation, the `extcodecopy` opcode does not mask the address before passing it to the `getRawCodeHash` function. While address masking is performed elsewhere (e.g., within the `$llvm_AlwaysInline_llvm$_warmAddress` function), it is not done in `getRawCodeHash`.

The `getRawCodeHash function` of the `AccountCodeStorage` contract is implemented in Solidity, where the compiler inserts argument validation checks to prevent invalid data from being passed. If invalid data is detected, the function reverts with zero bytes. This causes an unexpected state in the EVM emulator's `fetchFromSystemContract` function, ultimately terminating the EVM execution environment.

Consider applying address masking before invoking `getRawCodeHash` to ensure that all addresses conform to the expected 160-bit format, as is done in the implementation of other opcodes.

Update: *Resolved in pull request #1116 at commit 6a76d76.*

M-06 Calldata is Accessible During EVM Contract Construction

In the EVM, when a contract is created, the calldata is empty. As a result, `calldatasize` and related opcodes return zero during the construction phase. In the current EVM emulator implementation, there is no check to determine whether the current frame is a constructor. This leads to non-empty calldata being returned during contract construction, causing a divergence from standard EVM behavior.

Consider verifying the `isConstructing` flag in bytecode hash for the current account. If it has been set, mimic the EVM behavior.

Update: *Resolved in pull request #1160 at commit 684c072, pull request #1196 at commit 16b76a2. The Matter Labs team stated:*

> *Different opcode implementations were made for the corresponding cases.*

M-07 `calldataload`/`calldatacopy` Behavior Diverges From EVM

In the EVM, the `calldataload` and `calldatacopy` opcodes return zero bytes when accessing an index that exceeds the calldata length, including extremely large indexes such as the maximum value of a 256-bit unsigned integer. However, in EraVM, these opcodes trigger a panic error if the index is greater than `2^32-33`, as documented in the EraVM documentation. This divergence may cause unexpected behavior in EVM-like environments that rely on the standard EVM implementation.

The current implementation of `calldataload` and `calldatacopy` for the EVM emulator does not account for EraVM's behavior. This could lead to inconsistencies in handling edge cases.

In order to replicate EVM behavior, consider implementing a check for the accessed index in the EVM emulator.

Update: *Resolved in pull request #1097 at commit 12efa1c, pull request #1196 at commit 16b76a2.*

M-08 Underestimated Gas Cost for `MCOPY`

The `MCOPY` opcode copies memory data from one memory area to another. The total gas cost consists of two parts: a static part that charges 3 gas units independent of the size of the copied data, and a dynamic part that charges extra gas units in case the memory expands during this operation.

In the `EvmEmulatorLoops` template, the `MCOPY` implementation is missing the static gas cost part, as it only charges the gas needed to expand the memory.

Consider always charging the static gas cost of the `MCOPY` operation to remain equivalent to the corresponding EVM implementation.

Update: *Resolved in pull request #1081 at commit 44bd426. The Matter Labs team stated:*

> *The missing static gas cost has been added.*

M-09 Nonce Not Incremented When Deploying EVM Contracts at Existing Addresses

In an EVM environment, when a contract creation opcode (`create` or `create2`) is executed, the caller's nonce should be incremented — even if the contract cannot be deployed due to the target address already having associated code or nonce — if other checks are satisfied.

In the current implementation of the EVM emulator, when a `create*` opcode is triggered and the `_executeCreate` function calls `precreateEvmAccountFromEmulator`, the nonce is temporarily incremented. If the subsequent nonce and code hash checks fail, the call reverts, rolling back the nonce increment. However, the callers frame execution continues, leaving the caller account's nonce unchanged, which differs from the standard EVM behavior. The same applies to the `create2EVM` function of the `ContractDeployer`. If the constructor reverts during deployment, the deployment nonce will remain unchanged, allowing the contract to be deployed at the same address in the future.

Consider aligning the emulator's behavior with the standard EVM by ensuring that the caller's nonce is incremented regardless of whether the target address already has associated code or nonce.

Update: *Resolved in pull request #1127 at commit ef80122. The Matter Labs team stated:*

> *We changed the creation flow to match EVM.*

M-10 EVM-to-EVM Contract Interaction Allows for Gas Griefing Attacks

The EVM charges gas for executing operations, ensuring that transactions consume resources proportionally to their complexity. When emulating these calls, a conversion between native gas units (ergs) and EVM gas is applied. The emulator aims to follow the EVM's behavior closely, accounting for gas according to the EVM specification.

However, the EVM call may consume more ergs than the corresponding amount of EVM gas, which is used to track execution costs at the EVM level. While the emulator aligns EVM gas charges with the EVM's rules, underlying operations such as loading bytecode or running the emulator itself also consume ergs that are not reflected in the EVM gas usage. This discrepancy allows a caller to force a callee to spend more ergs than intended, creating a situation where gas griefing attacks are possible. As a result, contracts cannot accurately control or limit the total resources consumed during EVM-to-EVM calls, leading to potential DoS, when the nested call aborts the EVM environment.

Consider keeping closer track of the actual ergs consumed compared to the EVM gas left during EVM-to-EVM calls so that gas griefing attacks are controllable. For example, consider calculating the actual ergs spent during an EVM call and adjust the EVM gas accordingly. Another possible solution is to also limit the ergs passed to an EVM callee frame in accordance to the call's gas limit.

Update: *Acknowledged, not resolved. The Matter Labs team stated:*

> *The current implementation guarantees compliance with EVM gas rules, the violation of which can cause problems and create potential attack vectors. Additionally, passing the entire native gas to the callee frame helps smooth out the discrepancy between EVM and native gas. Since the entire EVM environment reverts in the event of an out-of-gas error, gas estimation will, in most cases, return the amount of native gas required for the transaction to execute correctly. At the same time, it's true that the current implementation can break assumptions related to the* `try-catch` *pattern, particularly if a fixed gas value is passed to the call. This should be clearly documented and communicated to developers. For most use cases, the current design is the best available option. Further optimizations of the emulator may reduce gas consumption and simplify interactions with the EVM environment.*

Low Severity

L-01 Loose Stack Overflow Check

In the `EvmEmulatorFunctions` template, the `pushStackItem` function pushes a new item into the stack. Prior to pushing the new item, it is ensured that the stack has not overflown by checking whether the current stack pointer is lower than the `BYTECODE_LEN_OFFSET`.

However, consider the case where the current stack pointer is pointing to the slot right before the `BYTECODE_LEN_OFFSET` slot. Then, when calling `pushStackItem`, the aforementioned overflow check will pass, thus allowing the new item to be inserted and the incremented stack pointer to point to the `BYTECODE_LEN_OFFSET` slot. Even if, as per the current design, the stack's head is not actually stored in the memory and so the `BYTECODE_LEN_OFFSET` is not overwritten by this operation, the stack's head is typically out of the stack's space. A similar check with the same impact is performed within the `pushStackCheck` function, with the difference being that more than one new items are involved.

Consider making the stack overflow check stricter in both `pushStackItem` and `pushStackCheck` functions. This is so that a new item is only allowed to be placed onto the stack if there is available stack space for it.

Update: *Resolved in pull request #1085 at commit 5db2a30. The Matter Labs team stated:*

> *A stricter check has been implemented.*

L-02 Reverting with Misleading Error in `EvmGasManager`

In the `$llvm_AlwaysInline_llvm$_onlyEvmSystemCall` function of the `EvmGasManager` contract, the `isSystemCall` flag determines whether the function is being invoked as a system call. When this flag is set to 0, indicating that it is not a system call, the contract reverts with the `CallerMustBeEvmContract` error. However, the expected behavior is to revert with the `SystemCallFlagRequired` error, as implemented in other contracts. This discrepancy could lead to confusion during debugging or when integrating with the contract, as the error message does not align with the root cause of the failure.

Consider reverting with the `SystemCallFlagRequired` error to ensure clarity and consistency across the codebase.

Update: *Resolved in pull request #1166 at commit 8349c91.*

L-03 `INVALID` Opcode Behavior Divergence in EraVM<>EVM Interaction

When the `Invalid` opcode is executed in the EVM Emulator, it triggers the `revertWithGas` function, which stores the remaining gas value (0 in the case of an invalid opcode) in memory and reverts with a 32-byte value indicating the remaining gas. This behavior aligns with the EVM<>EVM interaction, where the gas consumed by the `Invalid` opcode is burned and 0 is returned as remaining gas. However, for EraVM<>EVM interactions, the behavior diverges. The `Invalid` opcode in EraVM must mimic the behavior of `revert(0,0)`, but the emulator produces `revert(0,32)` instead, exhibiting an inconsistency.

Consider maintaining consistency in opcode behavior across emulated environments during cross-VM interactions. Alternatively, consider documenting the specific differences in behavior for improved clarity.

Update: *Resolved in pull request #1180 at commits 0397674, d8b49e5.*

L-04 Absence of EVM Contract Force Deployment Logic

The `forceDeployOnAddress` function of the `ContractDeployer` contract enables the forced deployment of contracts. However, it may not align with the construction of EVM-type contracts. In typical contract deployment scenarios, different initialization procedures are required based on the contract type, and EVM contracts rely on the `_constructEVMContract` function instead of the `_constructContract` function used in this context.

Consider updating the `forceDeployOnAddress` function to check for the contract type and invoke the appropriate construction function.

Update: *Resolved in pull request #1179 at commits 3f76fcc, 0f87cf9 and pull request #1196 at commit 0d6fec7. The Matter Labs team stated:*

> *We added EVM force deployment functionality.*

L-05 Incorrect `MAX_EVM_BYTECODE_LENGTH` in `Utils` Library

The maximum bytecode length for Ethereum contracts is 24,576 bytes, as defined by EIP-170. Several components in the codebase rely on this length restriction for validation and processing purposes.

In the current implementation, the `validateBytecodeAndChargeGas` function of the `EVMEmulator` contract correctly enforces the 24,576-byte limit as specified by EIP-170. However, the `publishEVMBytecode` function of the `KnownCodesStorage` contract calls the `Utils.hashEVMBytecode` function, which references a `MAX_EVM_BYTECODE_LENGTH` constant that has been set to `(2 ** 16) - 1` (or 65,535). This inconsistency introduces a mismatch between the actual EIP-170 constraint and the system's internal validation, potentially leading to the deployment of the contract with a bigger length than allowed or with invalid assumptions about related functionality.

Consider unifying the `MAX_EVM_BYTECODE_LENGTH` constant across the codebase to align with the EIP-170 standard while keeping a small buffer for potential bytecode padding to ensure consistent behavior.

Update: *Resolved in pull request #1181. The Matter Labs team stated:*

> *This check is used to prevent overflow of 2 bytes used to encode published bytecode length. Technically, it is allowed in EVM to have and call contracts bigger than 24,576 bytes (example of corresponding test). In our current EVM emulator design, it is not possible to execute bytecodes exceeding max allowed length, so we added corresponding check in the `getDeployedBytecode` function.*

L-06 Incorrect Custom Error Revert in `EvmGasManager`

The `$llvm_AlwaysInline_llvm$_onlyEvmSystemCall` function of the `EvmGasManager` contract is responsible for validating the current call. When certain conditions are not met, the function reverts using `revert(0, 32)`. According to the comments in the code, this revert is intended to represent the `CallerMustBeEvmContract()` custom error. In the EVM, a revert triggered by a custom error without arguments should produce an error payload of 4 bytes (the selector of the custom error). However, the current implementation incorrectly reverts with 32 bytes.

Consider modifying the revert logic to ensure that it reverts with only the 4-byte selector.

Update: *Resolved in pull request #1166 at commit 8349c91.*

L-07 EVM Emulator Uses `call` for `isSlotWarm` Function

The `isSlotWarm` function of the `EvmGasManager` contract does not perform any state-modifying operations, such as `tstore` or `sstore`. This could logically classify it as a `view` function. However, the EVM Emulator employs a `call` for this function, and the accompanying comment incorrectly states that the `call` is necessary due to the use of the `tstore` operation, which is not actually present in this function.

To increase protocol safety by preventing unintended state changes, consider replacing `call` with `staticcall` for the `isSlotWarm` function.

Update: *Resolved in pull request #1167 at commit f2677bf.*

L-08 Behavior Divergence with EVM for Create* Opcodes

According to the EVM specifications outlined in EIP-2929, during the execution of the Create/Create2 opcodes, the account being deployed must be warmed before performing checks and executing its initialization code, and it must stay warm afterward.

In the `_executeCreate` function of the `EvmEmulator` contract, if the `precreateEvmAccountFromEmulator` external call fails, the intended deployed account is never warmed. This diverges from EIP-2929, which mandates that the account be warmed prior to any validation steps and remain so during and after the initialization code execution. Similarly, during the creation of an EVM contract from the EraVM context, in the event of a construction revert, the contract being constructed remains in a cold state. However, it was warmed during the execution of EVM context and should remain warm.

In accordance with EIP-2929, consider warming the contract address before any validation steps are performed.

Update: *Partially resolved in pull request #1127 at commit ef80122. The Matter Labs team stated:*

> *Contract creation flow from the EVM emulator was changed to match EVM. Regarding the creation of a EVM contract from the EraVM context, at the moment we do not plan to make changes.*

L-09 Unexpected Balance Increase for EVM Contracts Without Fallback/Receive Functions

On Ethereum, there are scenarios where a contract's balance can increase without triggering its code, such as through `selfdestruct` or validator withdrawals using Eth1 withdrawal credentials. On ZKsync, `selfdestruct` cannot be executed. However, an operator-specified address can still receive funds directly—without invoking contract code—using the `directETHTransfer` function in the `Bootloader`, consistent with Ethereum's behavior.

Another possibility is when a paymaster sends excess funds and receives some portion back as a refund on the ensure payment step, or receives a refund for an L2 transaction. If the paymaster is an EVM contract, its balance can increase without code execution, potentially causing discrepancies in internal paymaster balance accounting.

Consider maintaining consistency with EVM or informing users about unexpected balance increases, ensuring that users do not rely on code execution to detect changes in the contract's balance.

Update: *Acknowledged, not resolved. The Matter Labs team stated:*

> *This behavior should be reflected in the documentation.*

L-10 Incorrect Gas Handling in `delegatecall` Implementation

The EVM emulator implements the `performDelegateCall` function, which allows one smart contract to execute code from another contract while maintaining the original contract's storage.

The EVM emulator prematurely charges gas for memory expansion and account access during a `delegatecall`, without first verifying whether the target account type. This results in gas charges even if the delegatecall does not proceed due to the target being non-EVM.

Consider verifying the target account type before executing charging operations.

Update: *Resolved in pull request #1120 at commit 3299faa, pull request #1196 at commit 0e0a7ac. The Matter Labs team stated:*

> *Delegatecall flow has been changed to better match EVM. Now delegatecalling the EraVM contract looks to the user just like calling a contract that immediately reverts.*

Notes & Additional Information

N-01 Hardcoded Memory Offsets

Memory slots 23-31 are reserved for cached fixed context values, such as `block.timestamp` or the gas limit of the transaction. The offsets for each of the values are hardcoded, which can be error-prone.

In order to enforce consistency and avoid mistakes, consider chaining the offsets one after another, as is done afterwards for subsequent sections in the memory layout.

Update: *Resolved in pull request #1168 at commit 34918c3.*

N-02 Naming Suggestions

In the `EvmEmulatorFunctions` template, multiple opportunities for improved naming were identified:

- The return variable of the `expandMemory2` function is named after "maxExpand", whereas the function returns the gas cost for the memory expansion.
- The parameters of the `checkMemIsAccessible` function are named `index` and `offset`, whereas all the call sites provide arguments named "offset" and "size", respectively.
- The `UINT32_MAX` function could be renamed to `MAX_UINT32` for consistency with the rest of the code in the contract.
- There is a typographical error in the `publishEvmBytecode` function of the `KnownCodesStorage` contract. The"vesionedBytecodeHash" local variable should be named "versionedBytecodeHash".

Consider renaming the variables mentioned above for improved readability and clarity.

Update: *Resolved in pull request #1169 at commit 7328b8c and in pull request #1082 at commit 8e8323d.*

N-03 Misleading Documentation

Throughout the codebase, multiple instances of misleading documentation were identified:

In the `EvmEmulatorFunctions` template:

- In line 345, there is a factual mistake: "bits" should be "bytes". The comment should also clearly state that it is referring to the first 32 bytes *of the* `returndata`.
- In line 932, the comment suggests that all the returndata is skipped, whereas only the initial 32 bytes of the returned gas value are skipped.

Consider addressing the above instances of misleading documentation to improve the readability of the codebase.

Update: *Resolved in pull request #1170 at commit ba441ac.*

N-04 Gas Optimizations

Throughout the codebase, multiple opportunities for gas optimization were identified:

- There are several instances where a value is divided with a divisor raised to a power of 2 using the `div` opcode. Consider using the cheaper `shr` instead. For example, `div(value, 512)` could be `shr(9, value)`.
- The implementation of the `CODECOPY` opcode checks for out-of-bounds bytecode access by comparing `sourceOffset` with `MEM_LEN_OFFSET`. Consider using the ending slot of the stored bytecode to improve code clarity and save gas costs by using the more efficient `$llvm_AlwaysInline_llvm$_memsetToZero` function for as many bytes as possible instead of the costlier `$llvm_AlwaysInline_llvm$_memcpy` function.

Consider adopting the above suggestions for optimized gas costs.

Update: *Resolved in pull request #1171 at commits dbfa843, 3f5866d and pull request #1156 at commits c621387, 624bf1c.*

N-05 Code Simplification

Throughout the codebase, multiple opportunities for code simplification were identified:

`EvmEmulatorLoop.yul`:

- The `OP_PC` opcode implementation calculates the current program counter (pc) in relation to `BYTECODE_OFFSET`. However, instead of immediately subtracting `BYTECODE_OFFSET` from `ip`, `BYTECODE_LEN_OFFSET + 32` is used which equals `BYTECODE_OFFSET`. Consider simplifying the code by using the `BYTECODE_OFFSET` value.
- All the instances of `PUSHX` have similar content. The code only differentiates in the X value, which is the size of bytes to read from the bytecode and increase the `ip` with. The content could be refactored into an `internal` function to avoid having the same number duplicated and charging the same amount of gas.

`Executor.sol`:

- In the `batchMetaParameters` function, the `s.l2DefaultAccountBytecodeHash` and `s.l2EvmEmulatorBytecodeHash` state variables can be directly passed in the encoding instead of unnecessarily assigning them to local variables first.

`Constants.sol`:

- Consider listing the system contracts in an ordered way based on their address with regard to `SYSTEM_CONTRACTS_OFFSET` for clarity. Specifically, `CODE_ORACLE_SYSTEM_CONTRACT` and `EVM_GAS_MANAGER` should be listed after `PUBDATA_CHUNK_PUBLISHER`.

Consider simplifying the code as suggested above in order to improve the overall readability and clarity of the codebase.

Update: *Resolved in pull request #1174 at commits 461e9c1, 405eff1, f3de6b3.*

N-06 Unused Functions

Having unused functions in the codebase can negatively affect code clarity and maintainability. The `MSG_VALUE_SIMULATOR_STIPEND_GAS` function is not invoked anywhere in the codebase, making its purpose unclear. Similarly, the `pushStackItemWithoutCheck` and `pushStackCheck` functions are also never used.

Consider removing any unused functions from the production code to improve code readability and maintainability. Alternatively, if such functions serve a specific purpose, ensure that they are used appropriately.

Update: *Resolved in pull request #1172 and in pull request #1085 at commit 5db2a30.*

N-07 Redundant Logic in `_constructEVMContract`

The `_constructEVMContract` function of the `ContractDeployer` contract includes a call to the `_storeConstructingByteCodeHashOnAddress` function using a dummy bytecode hash. In this process, the `isConstructing` bit is already set to `true` in the provided value. However, `_storeConstructingByteCodeHashOnAddress` clears the `isConstructing` bit, sets it back again, and subsequently invokes the `storeAccountConstructingCodeHash` function. This sequence involves unnecessary steps, as bit manipulation is redundant when the `isConstructing` bit is already pre-set.

Consider calling the `storeAccountConstructingCodeHash` function directly with the dummy bytecode hash. This avoids the redundant clearing and resetting of the `isConstructing` bit, thereby streamlining the logic and reducing gas usage.

Update: *Resolved in pull request #1173 at commit 5261075.*

N-08 Repetitive Logic During EVM-to-EVM Contract Creation

When an EVM contract deploys another EVM contract, the `precreateEvmAccountFromEmulator` function is invoked. This function ensures that it is allowed to deploy EVM contracts and that a contract with the same address has not been deployed previously. Subsequently, the EVM emulator calls `createEvmFromEmulator`, repeating the same checks, which consumes additional gas and unnecessarily increases the bytecode size of the `ContractDeployer` contract.

Consider streamlining the EVM-to-EVM contract deployment process to eliminate repeated logic and reduce unnecessary gas usage.

Update: *Resolved in pull request #1127 at commit 215aa6d.*

N-09 Magic Numbers

Throughout the codebase, multiple instances of magic numbers were identified:

- In `Utils.sol`, the `0xff` number is used instead of the `CREATE2_EVM_PREFIX` constant which is defined in `Constants.sol` but remains unused.
- In the `hashL2Bytecode` function of `Utils.sol`, the upper length limit of an EraVm bytecode is hardcoded, whereas the `MAX_EVM_BYTECODE_LENGTH` constant could be used instead.
- In `EvmEmulatorFunctions.template.yul`, the 160-bit address mask is used multiple times. Consider making it a constant for improved clarity.

Consider using constant variables instead of magic numbers throughout the codebase for enhanced clarity and readability.

Update: *Resolved in pull request #1175 at commits c712ced, 4c73482 and 4b1fc30.*

N-10 Missing Documentation

Throughout the codebase, multiple instances of missing documentation were identified:

- In `EvmEmulatorFunctions.template.yul`, there is an empty memory slot between `LAST_RETURNDATA_SIZE_OFFSET` and `STACK_OFFSET`. Consider documenting that the empty slot is used to denote an empty stack.
- In `EvmGasManager.yul`, the `$llvm_AlwaysInline_llvm$__getRawSenderCodeHash` function uses a low-level call with a function selector. Consider documenting the corresponding function's name.
- In `EvmEmulatorFunctions.template.yul`, the `copyRest`, `memCpy`, and `memsetToZero` helper functions use plenty of bitwise operations for efficiency. Consider providing documentation regarding the implementation logic of these functions.

For improved code clarity and maintainability, consider providing documentation for all parts of the codebase where the implementation logic is not high-level or straightforward enough.

Update: *Resolved in pull request #1176 at commits 1d8f871, 20f37ff and e24895c.*

N-11 `prevrandao` Opcode Implementation Inefficiency

According to the ZKsync documentation, the `prevrandao` value on the ZKsync network is constant and set to `2500000000000000`. However, the current implementation of the EVM Emulator performs several additional operations to retrieve this value from the `SystemContext` contract and then caches it in memory.

Consider pushing the constant value directly to the stack instead of making an external call to the `SystemContext` contract, provided there is no plan to introduce a configurable value for `prevrandao`. This would reduce the computational overhead and save ergs for users using the EVM emulator.

Update: *Resolved in pull request #1178 at commit 7f6e3f1.*

N-12 Redundant Warming of EVM Account During Construction

In the `EVMEmulator` contract, the account being constructed is explicitly warmed at the start of the constructor execution using the `$llvm_AlwaysInline_llvm$_warmAddress` function. However, this warming is unnecessary, as the account is warmed by other segments of the code depending on the deployment context.

For deployment in the EraVM context, the `consumeEvmFrame` function is called after the direct warming of the account, which also warms the caller (the constructed account in this scenario). Similarly, for deployment in the EVM context, the `$llvm_AlwaysInline_llvm$_warmAddress` function is invoked in the `_executeCreate` function before the deployment occurs. Thus, in all cases, the explicit warming of the constructed account within the constructor is redundant.

Consider removing the redundant warming logic to avoid unnecessary duplication.

Update: *Resolved in pull request #1177 at commit 65e8375.*

N-13 Free Deployment Of Evm Contract

It is possible to invoke the ContractDeployer's `createEVM` and `create2EVM` functions and deploy contracts while consuming zero EVM gas.

This may happen if the deployer provides any deployment code that begins with the `STOP` opcode, which consumes 0 gas. Furthermore, the caller can specify an erg amount for the call that is less than the required overhead. This bypasses the deployment checks and allows the contract deployment to proceed without consuming any EVM gas.

Consider reverting the execution frame in case the ergs passed for a contract construction call is less than the required overhead of 2000 ergs, if this is intended to be an invariant for the system.

Update: *Partially resolved in pull request #1196 at commit bbfd8e7. The Matter Labs team stated:*

> *Now we charge 32,000 gas if EVM contract is created by EOA or EraVM contract. This additional cost corresponds to the EVM behavior.*

N-14 Stipend Value is Added Twice to the Call's Gas

In the CALL implementation of the EVM Emulator, if the value is greater than zero, a stipend of 2300 EVM gas is applied. However, the same amount of native gas (ergs) is also added in the MsgValueSimulator to the gas amount as stipend.

Consider verifying that the observed behavior aligns with the expected protocol behaviour. If it does not, ensure proper gas accounting by adjusting or removing one of the gas stipends accordingly.

Update: *Acknowledged, not resolved. The Matter Labs team stated:*

> *This is expected behavior.*

Conclusion

The Evm Emulator contract has been introduced to the ZKsync network to support the interpreted execution of EVM bytecode on top of EraVM. The emulator is designed to follow the EVM execution environment rules as closely as possible. Perfect equivalence cannot be achieved, since the actual execution environment is EraVM. However, EVM contracts should be able to be deployed and executed with a seamless experience. The current implementation follows the specification after the Cancun fork. To support the changes of any subsequent upgrade forks, the emulator's code should be refactored.

The report highlights a number of issues, primarily concerning the functional correctness of EVM bytecode execution, equivalence with the EVM specification, and opportunities to improve code clarity and readability. Since the audited codebase operates at a very low level of contract execution, further in-depth testing is highly advised.

The Matter Labs team provided us with documentation regarding the EVM emulator design and has been very responsive throughout the audit duration.