Zellic

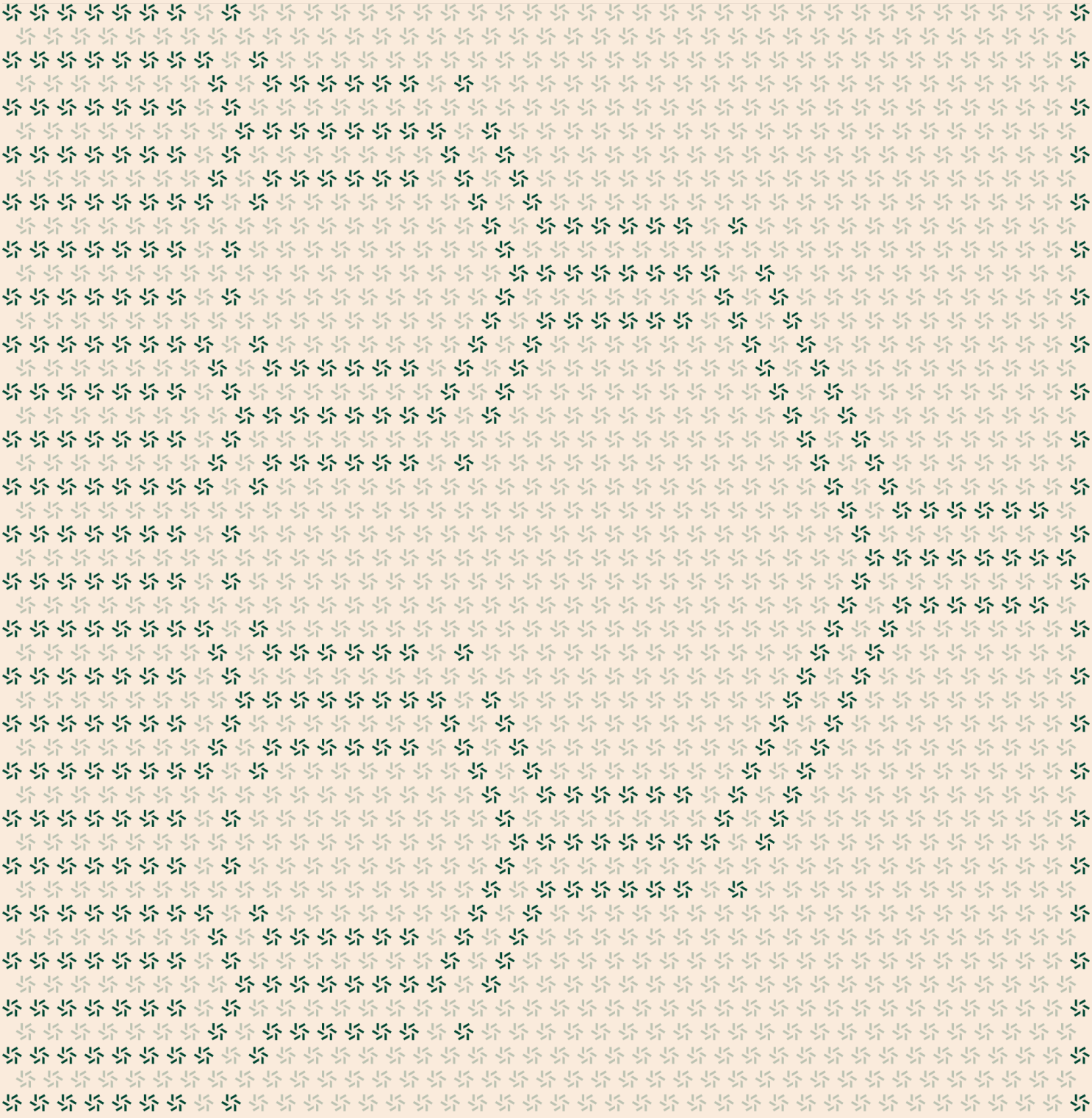
**Prepared for**
Clément Walter
KKRT Labs

**Prepared by**
Filippo Cremonese
Jinseo Kim
Zellic

**December 12, 2024**

# Kakarot zkEVM

## Diff Review



# Contents

## About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

# 1. Introduction

## 1.1. About Kakarot zkEVM

KKRT Labs contributed the following description of Kakarot zkEVM:

> Kakarot is a zkEVM built in Cairo, the provable language that powers Starknet and all the StarknetOS chains (also called CairoVM chains, or Starknet appchains).

## 1.2. Scope

The engagement involved a review of the following targets:

### Kakarot zkEVM Contracts

| | |
|---|---|
| **Types** | Solidity, Cairo0 |
| **Platforms** | EVM-compatible, Starknet |
| **Target** | kakarot |
| **Repository** | https://github.com/kkrt-labs/kakarot ↗ |
| **Version** | `672d3193ea63bdf5aedb358195565a55ad1a8411` |
| **Programs** | `(diff review)`<br>`solidity_contracts/src/Kakarot/Coinbase.sol`<br>`solidity_contracts/src/CairoPrecompiles/DualVmToken.sol`<br><br>`solidity_contracts/src/CairoPrecompiles/WhitelistedCallCairoLib.sol` |

| | |
|---|---|
| **Target** | kakarot-lib |
| **Repository** | https://github.com/kkrt-labs/kakarot-lib ↗ |
| **Version** | `a3714b332c6fb189615d770ddb89e271b43cf430` |
| **Programs** | `CairoLib` |

| | |
|---|---|
| **Target** | cairo-contracts |
| **Repository** | https://github.com/OpenZeppelin/cairo-contracts ↗ |
| **Version** | `1f9359219a92cdb1576f953db71ee993b8ef5f70` |
| **Programs** | `Ownable`<br>`Pausable` |

## Contact Information

The following project managers were associated with the engagement:

**Jacob Goreski**
Engagement Manager
jacob@zellic.io ↗

**Chad McDonald**
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

**Filippo Cremonese**
Engineer
fcremo@zellic.io ↗

**Jinseo Kim**
Engineer
jinseo@zellic.io ↗

### 1.3. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zel-

lic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

# 2. Detailed Findings

## 2.1. Gas accounting issue

| | | | |
|---|---|---|---|
| **Target** | kakarot_precompiles.cairo | | |
| **Category** | Coding Mistakes | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

### Description

The multicall precompile receives input in the form `number_of_calls||call1||call2||...`. However, even though the `number_of_calls` is read and used to compute the gas cost, the actual number of calls made depends only on the length of the input data:

```
func cairo_multicall_precompile{
    syscall_ptr: felt*,
    pedersen_ptr: HashBuiltin*,
    range_check_ptr,
    bitwise_ptr: BitwiseBuiltin*,
}(input_len: felt, input: felt*, caller_address: felt) -> (
    output_len: felt, output: felt*, gas_used: felt, reverted: felt
) {
    // [...]

    let number_of_calls = Helpers.bytes2_to_felt(number_of_calls_ptr + 30);
    let gas_cost = number_of_calls * CAIRO_PRECOMPILE_GAS;
    let calls_ptr = number_of_calls_ptr + NUMBER_OF_CALLS_BYTES;
    let calls_len = input_len - NUMBER_OF_CALLS_BYTES;

    let (output_len, output, reverted) =
    Internals.execute_multiple_cairo_calls(
        caller_address, calls_len, calls_ptr
    );
    return (output_len, output, gas_cost, reverted);
}
```

The `number_of_calls` is not provided to `Internals.execute_multiple_cairo_calls`.

### Impact

Users of the multicall precompile may provide a mismatching `number_of_calls` and be charged a higher- or lower-than-intended amount of gas.

This issue was initially classified as medium impact; after discussing with the Kakarot team, we agreed to lower its severity to low, since transaction relayers are ultimately responsible for ensuring any transactions they submit do not cause them a loss due to gas charges. Since the behavior reported in this issue is deterministic, it does not allow on its own to trick a relayer which is performing a transaction simulation to estimate gas charges into submitting an unprofitable transaction.

### Recommendations

Assert that the length of the input is consistent with the `number_of_calls` parameter.

### Remediation

This issue has been acknowledged by KKRT Labs, and fixes were implemented in the following commits:

- `3f7d131a` ↗
- `302775d7` ↗

### 2.2. Off-by-one overflow in `byteArrayToString`

| **Target** | CairoLib.sol | | |
|---|---|---|---|
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

#### Description

The `byteArrayToString` function in the CairoLib contract converts the given bytes representation of a `ByteArray` variable into the string variable in Solidity. The following is the structure of a `ByteArray` variable:

```
pub struct ByteArray {
    full_words_len: felt252,
    full_words: [<bytes31>],
    pending_word: felt252,
    pending_word_len: usize,
}
```

Because felt, the fundamental type in Cairo, can only store up to 251-bit data, each "word" in `ByteArray` stores 31-byte data. In the main loop used to copy data, the `byteArrayToString` function is concatenating these 31-byte words into a single bytes variable:

```
assembly {
    resultPtr := add(result, 32)
    // Copy full words. Because of the Cairo -> Solidity conversion,
    // each full word is 32 bytes long, but contains 31 bytes of information.
    for { let i := 0 } lt(i, fullWordsLength) { i := add(i, 1) } {
        let word := mload(fullWordsPtr)
        let storedWord := shl(8, word)
        mstore(resultPtr, storedWord)
        resultPtr := add(resultPtr, 31)
        fullWordsPtr := add(fullWordsPtr, 32)
    }
    // Copy pending word
    if iszero(eq(pendingWordLen, 0)) { mstore(resultPtr, shl(mul(sub(32,
    pendingWordLen), 8), pendingWord)) }
}
```

Note the way 31-byte data is written to memory. It is first shifted to the left by eight bits, then written to memory with the `mstore` instruction, which writes the given 32-byte data to the given memory

address.

This behavior allows the one byte, after the memory area to be written, to be overwritten to zero.

### Impact

We do not believe this finding can be practically exploitable or affect the behavior of the contract in a negative way, considering the memory-allocation algorithm used by the Solidity compiler. We also believe it to be unlikely for this behavior to change in the near future. However, relying on specific implementation details of a language implementation should be discouraged, especially if not strictly necessary, as it makes the code less maintainable and more likely for bugs to be introduced with possible future code changes.

### Recommendations

Consider copying data to memory with an exact size or document this behavior as intended.

### Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in commit `12a5b75e` ↗.

### 2.3. Version of the transaction is unnecessarily checked

| | | | |
|---|---|---|---|
| **Target** | account_contract.cairo | | |
| **Category** | Business Logic | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

#### Description

While reviewing the entry points for the new transaction flow, we observed that the `execute_from_outside` function asserts the version of the Starknet transaction to not be zero:

```
// EOA specific entrypoints
@external
func execute_from_outside{
    // (...)
}(
    // (...)
) -> (response_len: felt, response: felt*) {
    // (...)
    let (tx_info) = get_tx_info();
    let version = tx_info.version;
    with_attr error_message("Deprecated tx version: {version}") {
        assert_le(1, version);
    }
    // (...)
}
```

However, we do not see the reason for checking this. Typically, this check can be found on the `__execute__` function of SRC-6 contracts because the `__execute__` function assumes that the signature validation was finished on the `__validate__` function, but version-zero transactions can invoke any function as an entry point without executing the validation function. The `execute_from_outside` function does not assume that the validation is already finished but properly performs the signature validation in its library function:

```
func execute_from_outside{
    // (...)
}(...) -> (
    // (...)
) {
    // (...)
    Signature.verify_eth_signature_uint256(
```

```
            msg_hash=msg_hash,
            r=r,
            s=s,
            v=y_parity,
            eth_address=address,
            helpers_class=helpers_class,
        );
        // (...)
    }
```

### Impact

This finding does not constitute an exploitable security issue but a minor, unnecessary gas cost. It caught our attention and is reported since it is an unusual addition to the very critical codepath of the `execute_from_outside` function, in which we otherwise found no flaws.

### Recommendations

Consider not checking the transaction version in the `execute_from_outside` function.

### Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in commit `116b488f` ↗.

## 2.4. Incorrect interface

| | | | |
|---|---|---|---|
| **Target** | interfaces.cairo | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The interfaces of the contracts are listed in interfaces.cairo. Some of the interfaces are incorrect or missing:

- `Kakarot.set_authorized_pre_eip155_tx` receives the `msg_hash` parameter as `Uint256`, but the interface assumes its type as felt.
- `Kakarot.deploy_externally_owned_account` returns the `starknet_address`, but it is not reflected in the interface.
- `Kakarot.eth_call` receives the parameters `access_list_len` and `access_list`, but the interface does not include these parameters.
- The interface of `Kakarot.eth_estimate_gas` is missing.

### Impact

This finding does not affect the major logic of project, because none of these functions are expected to be systematically invoked by third-party contracts. Still, the interface file of the project acts as the documentation of the external functions of the contracts, and it is ideal to maintain its accuracy.

### Recommendations

Consider revising the interfaces to match them with the corresponding functions.

### Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in commit `d8f92671` ↗.

### 2.5. Static calls to precompiles may affect EVM-observable state

| | | | |
|---|---|---|---|
| **Target** | kakarot_precompiles.cairo | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

#### Description

The addition of non-whitelisted Cairo call and multicall precompiles introduced a new issue with the `STATICCALL` opcode. In standard EVMs, this opcode guarantees that no state change can be made by the contract being invoked. This assumption does not appear to hold for Kakarot at the moment. An EVM account could invoke the Cairo call or multicall precompiles via `STATICCALL`, and the Cairo call could have persistent changes to the underlying chain state. These changes could even be observable inside the EVM (i.e. if a token transfer occurs, and an instance of the DualVmToken exists for that token).

#### Impact

It is possible for a `STATICCALL` to make state changes on the underlying Cairo chain which are permanent and observable inside the EVM. While this can be shown via a POC, the real-world impact of this issue is difficult to evaluate and depends on the EVM contracts deployed on Kakarot. Therefore, this issue is reported as informational.

It is unclear if this issue could be exploitable in practice, or if it is more likely to amount to a harmless quirk. We believe it to pose a risk to some contracts which may expect and rely on `STATICCALL` being idempotent and not causing state changes, including DEXes and other DeFi protocols which could reasonably be expected to work with DualVmToken assets.

#### Recommendations

Forbid the use of `STATICCALL` to invoke precompiles that could cause state changes (possibly allowing only a set of allowlisted trusted callers if necessary).

#### Remediation

This issue has been acknowledged by KKRT Labs.

# 3. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

## 3.1. Diff review

This section discusses changes to the Cairo0 sources made between tag `code4rena-2024-09` (commit `7411a552`) and the most recent commit on the `main` branch at the time of the engagement (commit `672d3193`). We attempted to provide precise references to commits and pull requests (PRs) where possible, but we note that our review mostly considered the diffs between the two commits. Therefore, any changes introduced after `7411a552` that have been overwritten by subsequent commits may not be discussed. We also note that changes that remediated Code4rena findings (generally identifiable by containing "KGA" in the commit message) were out of the scope of this engagement. Minor changes that do not significantly alter the contract logic (such as minor refactors, comments, etc.) may also have been excluded for conciseness.

### Changes to chain-ID parameter

Prior to these changes, the Kakarot EVM chain ID matched the chain ID of the Starknet chain that Kakarot was running on. This tied Kakarot to Starknet-specific implementation details. It also did not allow to deploy multiple Kakarot instances on the same chain, as the two instances would have the same chain ID, allowing signature-replay attacks across the repeated deployments. Finally, Starknet chain IDs could potentially have a large value for their chain ID, which is problematic since several widespread wallets (including MetaMask) only support chain IDs up to a value of 4503599627370476. Other wallets have an even stricter limit, allowing chain IDs of up to four bytes (4294967296).

These issues were initially addressed by PR #1472, merged in commit `1b9d320b`, which modified the chain ID used by Kakarot. The chain ID was still derived from the Starknet chain ID but taken modulo the value of the `MAX_SAFE_CHAIN_ID` parameter, fixed to 4503599627370476.

The above PR was superseded by PR #1571, merged in commit `d4ae7b72`. This pull request transformed the Kakarot chain ID into a storage parameter. This latter PR also removed the `MAX_SAFE_CHAIN_ID` parameter and the modulo operation. Responsibility for setting an appropriately low chain ID is left to the deployer (and is adequately documented in the project main README). The chain ID of a Kakarot deployment is set by invoking the `initialize_chain_id` function, which is accessible only to the contract owner and can only be invoked once.

We note that this initialization pattern is potentially error-prone, as it is possible to deploy a Kakarot instance without invoking `initialize_chain_id`; until `initialize_chain_id` is invoked, the chain ID used by the chain will effectively be zero, and it is possible to process transactions before the chain ID is initialized. The chain ID is set by deployment scripts, lowering the likelihood of errors. Asserting the chain ID to be set (different from zero) could be an additional defense measure to guard against deployment errors.

### Kakarot precompiles' access-control refactor

Several commits improved and simplified the implementation of access control to the Kakarot Cairo precompiles exposed to smart contracts running in the Kakarot EVM.

As of commit `672d3193`, access control has been moved to a separate file, cairo_zero/kakarot/precompiles/precompiles_helpers.cairo. The `exec_precompile` function handling precompile execution invokes `PrecompilesHelpers.is_call_authorized` to determine whether the caller has permission to invoke the target precompile. Previously, this check was inlined and more difficult to review in isolation.

The logic applied by `PrecompilesHelpers.is_call_authorized` depends on the target contract. Some precompiles are subject to whitelisting and can only be called by a precise set of EVM senders maintained by Kakarot. This is currently the case of the messaging and whitelisted call precompiles. This restriction allows to assume that the EVM address initiating the call is trusted and therefore allows the Cairo contracts to make assumptions about the incoming message, such as the layout and content of the calldata. For example, this allows the messaging precompile to reliably determine the EVM address of the sender of an L2→L1 message. Only one specific EVM contract can directly invoke the Cairo messaging precompile; this contract exposes a public function that anyone can call, which includes the `msg.sender` into the calldata supplied to the Cairo precompile.

We note that the whitelisting mechanism uses a single global whitelist; this implies that any whitelisted EVM caller can invoke any precompile subject to whitelisting. This widens the attack surface more than strictly necessary, as compromising any whitelisted contract to gain the ability to make arbitrary calls would expose all whitelisted precompiles. However, it must also be noted that this solution allows to simplify the code and that maintaining individual access-control lists for each precompile could be more likely to be subject to mistakes.

Additionally, `PrecompilesHelpers.is_call_authorized` also prevents some precompiles from being called via `delegatecall` or `callcode` (even if not subject to caller whitelisting). Currently, the Cairo call and multicall precompiles are forbidden from being called using these two opcodes. This restriction is needed for security purposes, as these EVM opcodes can be used to execute bytecode from an arbitrary address while preserving the current `msg.sender` value, which could be exploited by malicious contracts to spoof calls if it were possible to use `delegatecall` or `callcode` to call these two precompiles.

### Call and multicall precompiles

PR #1503, merged in commit `44f25fdd`, and PR #1509, merged in commit `2b8ec8c7`, introduced two new precompiles, allowing any EVM account to perform a single Cairo call or multiple Cairo calls.

These precompiles are not subject to the whitelist authorization described above. The `PrecompilesHelpers.is_call_authorized` function does not check if the caller is whitelisted. Instead, it is checked that these precompiles are not invoked using `delegatecall` or `callcode` instructions. This effectively allows the accounts to freely interact with the external Cairo contracts.

In order to implement the multicall Cairo precompile, the parsing and Cairo call-execution logic of the `KakarotPrecompiles.cairo_precompile` function was modularized to the

`parse_cairo_call` and `execute_cairo_call` functions, and the `multicall_cairo_precompile` and `execute_multiple_cairo_calls` functions were created.

The multicall Cairo precompile receives the number of calls to be executed, followed by the concatenation of encoded calls. The number of calls to be executed is used to deduct the Cairo precompile gas, which is charged per call. One of the return values of the `parse_cairo_call` function is `next_call_offset`, which is effectively the total length of the parsed call. This value is ignored in the `cairo_precompile`, but the `execute_multiple_cairo_calls` function uses this value to find the offset of the next call to be executed. This is repeated until all of the given input is consumed through the parsing process. It is possible that the given number of calls does not match with the actual number of calls; see Finding 2.1. ↗.

It is worth noting that the maximum number of Cairo calls is limited to 65,535.

The implementation of the new Cairo precompile is equivalent to the original Cairo precompile, which can be only called by whitelisted contracts. The only difference between the Cairo precompile and the whitelisted Cairo precompile comes from the check in the `is_call_authorized` function.

### Fix `returndata` of create transactions

PR #1557, merged in commit `e3156b4c` ↗, fixes an incorrect behavior when processing contract-creation transactions. In those transactions, the stack is populated with incorrect data (the Starknet address and the EVM address of the deployed contract), instead of the bytecode of the deployed contract as returned by the contract constructor.

## 3.2. External Cairo call-flow refactor review

The flow used to invoke external Cairo contracts has had significant changes since the code revision reviewed during the previous Zellic audit (commit `464d254f` ↗).[1]

Previously, external Cairo contracts were invoked by the core Kakarot contract. This flow was changed so that external Cairo calls are now made by the account contracts. Since the sender of the Cairo call is now the account contract, precompile calls can now interact with preexisting Cairo contracts and use the authorization tied to the specific account contract that represents the EVM sender. In practice, this is useful — for example, performing a transfer of Cairo native ERC-20 tokens.

```
sequenceDiagram
    actor U as User
    participant KA as Kakarot Account
    participant KC as Kakarot Core
```

[1] The changes were introduced over multiple commits, which made it impractical to isolate them to individual commits. These observations were made by analyzing the relevant differences between commit `464d254f` ↗ and commit `672d3193` ↗.

```
    participant EC as External Cairo contract

    U -->> KA: execute_from_outside

    note over KA: Transaction validation

    KA ->> KC: eth_send_raw_unsigned_tx

    note over KC: Transaction execution invokes<br>CAIRO_CALL or
    CAIRO_MULTICALL precompiles

    note over KC: Precompile authorization checks

    KC ->> KA: execute_starknet_call

    KA ->> EC: Call contract
    EC ->> KA: (result)

    KA ->> KC: (result)

    note over KC: Transaction execution resumes
```

We note that due to changes made in commit 6bb28095 ↗, this mechanism cannot be used to reenter the Kakarot interpreter, as a reentrancy lock was added to the code `eth_call` function, preventing any possible reentrancy path into the interpreter.

### Potential DOS issue

This change introduces a potentially noteworthy interaction between Kakarot accounts and the ERC-20 used as EVM native currency; when an account contract is deployed and its initialization code is run, it invokes the Cairo ERC-20 used as the EVM native currency to grant infinite approval to the core contract. It is now possible for an account to invoke the ERC-20 and revoke the authorization. This prevents the core contract from transferring funds from the account, causing reverts in cases where the core contract uses `tranferFrom` to, for example, charge gas fees or commit a transfer.

This does not constitute a directly exploitable security issue, as the call can only happen with authorization from the corresponding account. However, especially in the case of smart contracts, this could potentially be abused by an attacker, if they were to be able to cause a victim smart contract to invoke the Cairo ERC20 via the call/multicall precompiles or through the DualVmToken contract.

> **Note:** The Kakarot team commented that they do not intend to deploy a DualVMToken instance that allows to interact with the native currency token. In PR #1622, merged in commit `6fbdc164` ↗, they also modified the DualVmToken contract to forbid calling `approve` using the core Kakarot address as a beneficiary for any token. This prevents using the DualVmToken contract to exploit this issue, but it does not address the attack surface exposed via call/multicall.

## 3.3. OpenZeppelin Ownable review

The Ownable library provides boilerplate that allows to implement simple access control, where a single sender is designated as the owner of a contract. This pattern is often used to implement access control for administrative actions. Note that while this library supports defining only a single owner, the owner address could be a contract with arbitrary logic (e.g., a multi-sig or a DAO).

We found no security issues in this library. The contract provides a simple interface, documented in the following sections.

### Function: `initializer(owner: felt)`

This function is intended to be used by the contract constructor, and it initializes the owner to the provided address by invoking `_transfer_ownership`.

Note that this function does not check whether the contract was previously initialized. We do not consider this a design choice that, although potentially dangerous, could constitute an exploitable issue on its own. Contracts using the Ownable library must not assume that calls to `initializer` will fail if the Ownable library has already been initialized.

The contract documentation specifies that the function must only be called once, without elaborating further on the potential consequences.

### Function: `_transfer_ownership(new_owner: felt)`

This function performs an ownership transfer, designating the specified address as the new owner of the contract. The transfer is performed unconditionally without performing any checks, which are a responsibility of the caller. This function is used by all other functions that perform an ownership transfer.

The documentation specifies that the function does not perform any access restriction.

We note that this function allows to transfer ownership to any address, including any invalid address. The Ownable library does not implement two-step transfers, where the recipient of the ownership role must invoke a function to confirm the address exists. While this opens the possibility of user errors, we consider this a design choice rather than a security issue. We acknowledge that more recent versions of the Ownable library have implemented two-step ownership transfer.

#### Function: `transfer_ownership(new_owner: felt)`

This function performs ownership transfer with access control, requiring the caller to be the current owner of the contract. The function also asserts that the address of the new owner is not the zero address.

#### Function: `assert_only_owner()`

This function must be called by functions of the contract using the library that should only be used by the owner. It asserts that the caller is the currently designated contract owner.

#### Function: `owner() -> (owner: felt)`

This function returns the address of the currently designated contract owner.

#### Function: `renounce_ownership()`

This function can be used to renounce ownership of the contract. It can only be called by the currently designated contract owner.

### 3.4. OpenZeppelin Pausable review

The Pausable library provides boilerplate to enable stopping contracts, commonly for emergency situations.

It defines one storage variable, which stores the state of the contract (0 meaning unpaused, any other value meaning paused). Contracts using the Pausable library must 1) invoke the functions `assert_not_paused` and `assert_paused` where necessary to assert the state of the contract and 2) expose functionality to pause/unpause the contract (by calling the library functions `_pause`/`_unpause`) after enforcing appropriate access control.

The library emits an event when the `_pause`/`_unpause` functions are invoked, recording the action and the address of the caller that performed it.

We found no issues in this library.

### 3.5. Solidity contracts review

This section discusses the role and logical structure of the Solidity contracts reviewed.

#### Coinbase

This contract is intended to be configured as the coinbase address of Kakarot.

The coinbase address of the Kakarot EVM is retrieved from the global variable of the core contract, which can be changed by the owner of the core contract. In Kakarot EVM, there are two aspects related to the handling of the coinbase address:

1. The gas spent for executing the transaction is transferred to the coinbase address.
2. The `COINBASE` instruction in EVM pushes the configured coinbase address into the stack.

Because of the first aspect, the coinbase address would receive native tokens. The Coinbase contract allows the owner of this contract to withdraw this token through the `withdraw` function.

The `withdraw(uint256 toStarknetAddress)` function transfers the whole balance of the received native token to the given address. Specifically, it 1) checks if the caller is the owner of the contract, 2) fetches the current balance of the contract by executing `address(this).balance`, and 3) invokes the Cairo call `token_address_dispatcher.transfer(toStarknetAddress, current_balance)`. The token address is preconfigured in the constructor and cannot be changed further. We believe it is reasonable because the native token address in the core contract is also preconfigured in the constructor and cannot be changed further.

The contract inherits the Ownable2Step contract, which implements the ownership mechanism as a two-step ownership-transfer mechanism.

### CairoLib and WhitelistedCallCairoLib

The CairoLib and WhitelistedCallCairoLib libraries provide the interfaces access to the Cairo precompile, Cairo multicall precompile, and Cairo whitelisted precompile. All functions in these libraries are marked as internal functions.

Both libraries include the `callCairo` and `staticcallCairo` functions. These functions receive the contract address, function selector, and calldata as parameters; invoke the Cairo precompile or whitelisted Cairo precompile; and then return the result. Note that the `staticcallCairo` function can modify the state of the external Cairo contract, because Starknet does not support invoking a function guaranteeing the state is unchanged. See Finding 2.5. ↗ for a more in-depth discussion of this.

There are `multicallCairo` and `multicallCairoStatic` functions in the CairoLib library, which are responsible to receive/encode multiple calls and invoke the multicall Cairo precompile. These functions do not exist in WhitelistedCallCairoLib since the whitelisted Cairo precompile does not support the multicall feature.

The WhitelistedCallCairoLib contract has the `delegatecallCairo` function, which invokes the whitelisted Cairo precompile with `DELEGATECALL`. Because the whitelisted Cairo precompile invokes the Cairo call on behalf of the account contract of the caller, and the context created by `DELEGATECALL` inherits the caller from the parent context, the Cairo call will be invoked on behalf of the account contract of the caller of the contract that invoked `delegatecallCairo`.

The CairoLib library includes the `byteArrayToString` function. This function decodes the given bytes representation of a `ByteArray` into a string variable, which can be useful to decode the string

returned by the Cairo contract since `ByteArray` is generally used in Cairo to handle the string type.

### DualVmToken

The DualVmToken contract allows the Kakarot accounts to seamlessly use Cairo ERC-20 tokens on the Kakarot EVM, using the whitelisted Cairo precompile. Specifically, this contract implements the standard ERC-20 interface and invokes `delegatecallCairo` to the Cairo ERC20 contract, which lets the caller account contract invoke the corresponding function of the Cairo ERC20 contract.

For convenience, the DualVmToken contract implements a variant of the ERC-20 interface that operates on Starknet addresses instead of on EVM addresses. This is achieved by exposing function overloads that receive `uint256` inputs instead of `addresses` (e.g., `balanceOf(uint256 starknetAddress)`). In fact, the standard ERC-20 functions that receive 20-byte EVM addresses operate by fetching the Starknet address of the EVM address through the `Kakarot.get_starknet_address` function.

## 3.6. Code4rena patches review

This section briefly discusses most changes remediating issues found during the Code4rena competition. Please note that reviewing these was not part of the scope of the engagement; this list is made for informational purposes, and no claim is made about the validity of the issues or the effectiveness of the remediations.

### Handling of base fees (KGA-34)

Kakarot was not correctly handling EVM base fees, allowing base fees to change in the middle of a block; the code was updated to ensure changes to the base fee only take effect at the start of a block.

Commit `cbe1c816` ↗ fixed this issue. The changes involved multiple files; the updated base fee is stored together with the block number from which it should take effect. The fee getters and setters were updated to account for the additional logic required to determine which fee parameter should be applied for the current block.

### Codehash commitment (KGA-80 and KGA-82)

Commit `d5932d4c` ↗ fixed an edge case where the codehash of a new account was returned incorrectly.

### Potential reentrancy issue (KGA-4, KGA-118, KGA-119, KGA-127, and KGA-133)

Commit `6bb28095` ↗ fixed a potential reentrancy issue. Kakarot is not reentrancy safe at the Cairo level. Previously, reentrancy checks were enforced at the account level (specifically, in the `execute_starknet_call` account-contract function). This did not prevent Cairo precompiles from reen-

tering the Kakarot EVM using a different account. The commit moved the reentrancy check to the `eth_call` function of the core Kakarot contract, making the reentrancy lock independent of the account initiating a Kakarot transaction.

### `execute_from_outside` function might return an incorrect `return_data_len` (KGA-72)

Commit `5cdbb52e` ↗ fixed an issue where in some cases `execute_from_outside` might have returned an incorrect length for the return data. The issue has unspecified impact and was deemed informational.

### Incorrect behavior of `BLOCKHASH` for the last 10 blocks (KGA-32, KGA-143, KGA-86, and KGA-88)

According to Kakarot documentation, the `BLOCKHASH` opcode should behave differently on Kakarot EVM; specifically, the block hash of the last 10 blocks cannot be obtained, and the `BLOCKHASH` opcode should return zero.

The handler for the opcode was not behaving according to the specification, causing a revert instead of returning zero for some of the previous 10 blocks.

Commit `d962582d` ↗ fixed the issue by adjusting a range check.

### Incorrect code address for deployment code (KGA-25 and KGA-73)

Commit `b2f53bad` ↗ addressed an issue where, while running contract-creation code during a deployment transaction, Kakarot was not using the correct code address for the deployment code.

### Dead code in `exec_create` (KGA-64)

Commit `36f8cdc6` ↗ removed some unreachable code in `exec_create`, which checked the return value of a call to `add_transfer` to charge the message value to the account that sent the message. The check is not needed since it is performed earlier.

### Gas-calculation underflow in `handle_l1_message` (KGA-70)

When handling L1 messages, Kakarot hardcodes a gas limit of 2.1B and a gas price of 1. The balance of the sender of the message is charged the maximum gas-fee amount at the beginning of the transaction. Unlike regular transactions processed through `eth_send_raw_unsigned_tx`, transactions submitted through `handle_l1_message` skipped a check ensuring the sender has sufficient balance to pay for the maximum gas fee, resulting in an underflow.

Commit `e791ac10` ↗ fixes the issue by setting the gas price to zero; the rationale behind this change

is that Starknet gas fees are paid by the L1 sender.

### Insufficient parameter validation in `ecrecover`

Commit `26fe525f` ↗ added missing validation of the `r` and `s` parameters to the `ecrecover` function.

Kakarot had not required these values to have a value lower than the secp256k1 module. This led to Kakarot accepting signatures that Ethereum considers invalid. Note that this signature-malleability attack does not allow to forge signatures for arbitrary payloads, but it does break an important invariant that could be relied on for security-critical purposes.

### RIPEMD160 off-by-one issue (KGA-18)

Commit `0a885b20` ↗ addressed an off-by-one issue in the RIPEMD160 implementation.

### RIPEMD160 missing dict finalization (KGA-14, KGA-28, KGA-74, KGA-132, and KGA-92)

Commit `b8853b47` ↗ added missing a call to `default_dict_finalize` in the RIPEMD160 implementation.

### Underconstrained loop exit condition in `felt_to_bytes_little` (KGA-31)

PR #1573 fixes an issue with `felt_to_bytes_little` which would allow a malicious prover to cause the function to return an incorrect value.

### Improper default dict finalization (KGA-39)

PR #1592, merged in commit `474951c7` ↗, fixed an issue where a default dict was not properly finalized before being copied.

### Chain-ID overflow in transaction decoding

PR #1581, merged in commit `158ff0c9` ↗, fixed an overflow that could occur while parsing chain IDs from legacy EIP-155 transactions.

### Useless computation in `uint256_fast_exp`

PR #1603, merged in commit `8eacec66` ↗, removed a useless, redundant recursive call of `uint256_fast_exp`, improving the contract efficiency.

### High-offset memory-expansion calculation (KGA-87)

PR #1604, merged in commit 289c2e89 ↗, fixed an arithmetic overflow that could occur when computing the gas cost of an operation that involves an expansion of the EVM-allocated memory.