Report

v. 1.0

Customer

StarkWare

Cryptography Audit

OS

21st November 2024

Report prepared by

ABDK Consulting

Contents

1 Changelog

#	Date	Author	Description
0.1	21.11.24	A. Zveryanskaya	Initial Draft
0.2	21.11.24	A. Zveryanskaya	Minor revision
1.0	21.11.24	A. Zveryanskaya	Release

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting (Mikhail Vladimirov and Dmitry Khovratovich) at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

StarkWare Industries is an Israeli software company that specializes in cryptography. It develops zero-knowledge proof technology that compresses information to address the scalability problem of the blockchain, and works on the Ethereum platform.

3 Project scope

We were asked to review the update to the Cairo library.

Files:

cairo/builtin_keccak/		
keccak.cairo		
cairo/builtin_poseidon/		
poseidon.cairo		
cairo/cairo_keccak/		
keccak.cairo		
cairo/cairo_secp/		
constants.cairo	ec.cairo	signature.cairo
field.cairo	bigint.cairo	
cairo/keccak_utils/		
keccak_utils.cairo		
cairo/		
cairo_builtins.cairo	hash_chain.cairo	hash_state _poseidon.cairo
hash_state.cairo	patricia_utils.cairo	patricia_with _poseidon.cairo
patricia.cairo	poseidon_state.cairo	sponge_as_hash.cairo
uint256.cairo		
starknet/contract_class/		
compiled_class.cairo	contract_class.cairo	
starknet/execution/		
deprecated _execute_syscalls.cairo	execute _entry_point.cairo	execute_syscalls.cairo
execute _transactions.cairo		

starknet/		
block_context.cairo	builtins.cairo	constants.cairo
contracts.cairo	os.cairo	output.cairo
state.cairo	transactions.cairo	

4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check if code actually does what it is supposed to do, if that algorithms are optimal and correct, and if proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Minor issues** contain code style, best practices and other recommendations.

5 Our findings

We provided the Client with a few recommendations.

Moderate	Info **14**	Fixed **5**
Minor	Info **8**	Fixed **8**

6 Moderate Issues

CVF-1. INFO

- **Category** Unclear behavior
- **Source** execute_syscalls.cairo

Description It seems that reasons spanning several felts are supported in the "FailureReason" structure, but this code only allows reasons one felt long.

Recommendation Consider either fully supporting reasons of multiple felts or removing support for them from the structure.

Client Comment *At the moment failed executions don't reach the OS, and are charged in a trusted manner (only check is that max_fee >= actual_fee). Thus, we never really handle failed executions in the OS curretly, so this is only partially implemented to support a 1 felt failure reason. Before we fully enable it, and pass through the OS for failed executions, we will handle failure reasons properly.*

```
1080 +// Write the failure reason.
1081 +tempvar start = failure_reason.start;
1082 +assert start[0] = failure_felt;
1083 +assert failure_reason.end = start + 1;
```

CVF-2. FIXED

- **Category** Unclear behavior
- **Source** execute_transactions.cairo

Description This limits the maximum fee by 2^128.

Recommendation Consider either removing this limitation or clearly stating it.

Client Comment *We will add a comment that the fee is limited by 2**128.*

```
249 +amount=Uint256(low=nondet %{ execution_helper.tx_execution_info.
    ↪ actual_fee %}, high=0),
```

CVF-3. FIXED

- **Category** Unclear behavior
- **Source** compiled_class.cairo

Description It is unclear how exactly builtins are encoded and how they are ordered.

Recommendation Consider clarifying.

Client Comment *We will add a comment that describes the expected order of builtins (the Cairo 0 compiler didn't compile when the order was wrong). This is relied upon in select_builtins.cairo, and checked (for Cairo >= 1) in the Sierra to CASM compiler.*

```
22 +// 'builtin_list' is a continuous memory segment containing the
      ↪ ASCII encoding of the (ordered)
```

CVF-4. INFO

- **Category** Suboptimal
- **Source** compiled_class.cairo

Recommendation As the compiled class version is a constant, and it is the first value hashed, then the hash state after it could be precomputed.

Client Comment *Here Poseidon is used, so the element is added to a list, and eventually we'll do length/2 hades permutations. It's not clear what to precompute in this case (unlike the older Pedersen case). A single hash in the entire OS execution is not very concerning.*

```
86 +hash_update_single(item=compiled_class.compiled_class_version);
```

CVF-5. INFO

- **Category** Suboptimal
- **Source** contract_class.cairo

Recommendation As the contract class version is a constant, and it is the first value hashed, then the hash state after it could be precomputed.

Client Comment *Same as above.*

```
50 +hash_update_single(item=contract_class.contract_class_version);
```

CVF-6. INFO

- **Category** Documentation
- **Source** state.cairo

Description The description of the arguments taken and the actual argument doesn't seems to match each other.

Recommendation Consider describing individual arguments in more details.

Client Comment *class_changes is a mapping class_hash → compiled_class_hash for declares within that block, hashed_class_changes is a mapping that is filled throughout the function from class_hash to the actual leaf content in the classes tree. I think it matches the current description, we'll try to make the comment clearer.*

```
257 +// Takes a dict mapping class hash to compiled class hash and
    ↪ produces
```

```
261 +    n_class_updates: felt, class_changes: DictAccess*,
    ↪ hashed_class_changes: DictAccess*
```

CVF-7. INFO

- **Category** Documentation
- **Source** keccak.cairo

Description The semantics of 'n_bytes' parameter is unclear and may lead to errors.

Recommendation Consider specifying,

```
129 +inputs: felt*, n_bytes: felt
```

```
140 +inputs: felt*, n_bytes: felt
```

```
149 +inputs: felt*, n_bytes: felt
```

CVF-8. INFO

- **Category** Suboptimal
- **Source** keccak.cairo

Description This function makes 44 128-bit range checks, which is a huge number for processing 1088 bits of data.

Recommendation It would be much more efficient to make inputs[] array of 128-bit numbers, range-check them with 10 checks, and then split into two parts each, spending at most 20 more checks. Even better way is to check smaller chunks with dedicated check functions.

Client Comment *These rangechecks are negligible compared to the keccak builtin. We may optimize in the future.*

```
161 +func _prepare_full_block{range_check_ptr, bitwise_ptr:
    ↪ BitwiseBuiltin*, keccak_ptr: KeccakBuiltin*}(
```

CVF-9. INFO

- **Category** Suboptimal
- **Source** hash_state_poseidon.cairo

Recommendation It would probably be more efficient to directly feed inputs into sponge, rather than accumulating them in memory first.

Client Comment *We considered it in the past but reached the opposite conclusion. Feeding it directly to the sponge would force us to pass the builtin_ptr and keep track of the parity of the # of arguments, which turns out to be more expensive.*

```
52 +let (hash) = poseidon_hash_many(n=hash_state.end - hash_state.start
   ↪ , elements=hash_state.start);
```

CVF-10. FIXED

- **Category** Procedural
- **Source** sponge_as_hash.cairo

Description The naming is inconsistent in many aspects: 1. Inputs use different letters ("x" and "y"), while outputs use indexes: ("result", "result1"). 2. Only one of two outputs has index. 3. Naming for capacity input and output differs from naming for other fields.

Recommendation A consistent naming would be, for example: x_in, y_in, c_in, x_out, y_out, c_out

Client Comment *Will consider renaming the stuct members (elsewhere they are called s0,s1,s2).*

```
8   +x: felt,
9   +y: felt,
10  +c_in: felt,
11  +result: felt,
12  +result1: felt,
13  +c_out: felt,
```

CVF-11. INFO

- **Category** Unclear behavior
- **Source** execute_syscalls.cairo

Description There is no range check for the "request..y_parity".

Recommendation Consider implementing an appropriate check.

Client Comment *It is checked inside try_get_point_from_x (assert_nn). With regards to completeness, in Sierra this syscall expects bool which is guranteed to be 0 or 1.*

```
981 +let (is_on_curve) = try_get_point_from_x(x=x, v=request.y_parity,
       ↪ result=response.ec_point);
```

CVF-12. INFO

- **Category** Unclear behavior
- **Source** execute_syscalls.cairo

Description There are no range checks for these values.

Recommendation Consider implementing appropriate checks.

Client Comment *Guranteed by Sierra, the inputs to these syscalls is u256, which is two u128 that are guranteed to be of the apprpriate size (execute_syscalls.cairo only deals with Cairo >= 1 syscalls).*

```
1002 +let (x) = bigint_to_uint256(ec_point.x);
1003 +let (y) = bigint_to_uint256(ec_point.y);
```

CVF-13. FIXED

- **Category** Unclear behavior
- **Source** execute_transactions.cairo

Description This logic allows one to avoid paying fee just by setting max_fee to zero.

Client Comment *This is used in testing, will be removed in the future.*

```
237 +if (max_fee == 0) {
238 +    return ();
239 +}
```

CVF-14. INFO

- **Category** Unclear behavior
- **Source** compiled_class.cairo

Description This function doesn't verify that "builtin_list" is properly ordered and contains only valid builtins.

Client Comment *True. compiled_class.cairo deals with the image of the Sierra→CASM compiler, which checks the values and order of the builtins. ATM compilation is handled by the sequencer (user signs the compiled artifcat), in the future we may have "full Sierra", which means verifying this compilation in the OS.*

```
63 +func validate_entry_points_inner{range_check_ptr}(
```

CVF-15. INFO

- **Category** Documentation
- **Source** compiled_class.cairo

Description The hashing is done using some tree-like structure in order to get different hashes for different classes. It is not clear though how different classes can be.

Recommendation Consider writing explicitly which kind of classes is supported.

Client Comment *This code computes the hash of a single compiled class, there are no different types (unless you consider deprecated_compiled_class, which represents cairo 0). Will add to the starknet docs how are compiled classes hashed, does it answer the question?*

```
84 +let hash_state: HashState = hash_init();
```

CVF-16. INFO

- **Category** Procedural
- **Source** builtins.cairo

Description The function doesn't actually implement this logic.

```
97 +// For the non-selected builtins (that is, selectable builtins that
      ↪  do not appear in
98 +// `selected_encodings`), this function validates that the
      ↪ difference is nonnegative
99 +// (that is, they weren't moved backward) and divisible by the
      ↪ builtin size.
```

CVF-17. INFO

- **Category** Suboptimal
- **Source** signature.cairo

Description Here 'verify_zero()' is called several times internally.

Recommendation Consider calling it directly instead.

```
147 +let (reduced_diff) = reduce(diff);
```

```
149 +return is_zero(reduced_diff);
```

CVF-18. INFO

- **Category** Flaw
- **Source** keccak.cairo

Description This condition is not checked with explicit constraints, and is only implicitly checked.

Recommendation Consider checking explcitly.

Client Comment *The only way to exploit this is instead of doing n_bytes/rate number of iterations, we can do (n_bytes+prime)/rate, which is impossible in practice.*

```
358 +if (nondet %{ ids.n_bytes >= ids.KECCAK_FULL_RATE_IN_BYTES %} != 0)
       ↪  {
```

CVF-19. FIXED

- **Category** Procedural
- **Source** sponge_as_hash.cairo

Description The capacity initialization constants are scattered across files and may collide.

Recommendation Consider defining them in a single file.

Client Comment *Will consider initializing to "2" in one place.*

```
3 +// c_in - the capacity part of the input (must be initialized to a
     ↪ constant, we use 2).
```

7 Minor Issues

CVF-20. FIXED

- **Category** Documentation
- **Source** compiled_class.cairo

Recommendation Consider explaining why there are 5 nonzero entries.

Client Comment *Will add a comment that explains it (currently the Cairo >= 1 gas mechanism is not in use).*

```
172 +assert builtin_costs[0] = 0;
173 +assert builtin_costs[1] = 0;
174 +assert builtin_costs[2] = 0;
175 +assert builtin_costs[3] = 0;
176 +assert builtin_costs[4] = 0;
```

CVF-21. INFO

- **Category** Bad datatype
- **Source** constants.cairo

Recommendation The numeric quotients used in these expressions should be named constants.

Client Comment *numerics only multiply # of steps or # of range checks, with dedicated constants the code will look worse (we'll need consts per each syscall).*

```
74  +const ENTRY_POINT_GAS_COST = ENTRY_POINT_INITIAL_BUDGET + 500 *
       ↪ STEP_GAS_COST;

76  +const FEE_TRANSFER_GAS_COST = ENTRY_POINT_GAS_COST + 100 *
       ↪ STEP_GAS_COST;

79  +const TRANSACTION_GAS_COST = (2 * ENTRY_POINT_GAS_COST) +
       ↪ FEE_TRANSFER_GAS_COST + (

83  +const CALL_CONTRACT_GAS_COST = SYSCALL_BASE_GAS_COST + 10 *
       ↪ STEP_GAS_COST + ENTRY_POINT_GAS_COST;
84  +const DEPLOY_GAS_COST = SYSCALL_BASE_GAS_COST + 200 * STEP_GAS_COST
       ↪  + ENTRY_POINT_GAS_COST;
85  +const GET_BLOCK_HASH_GAS_COST = SYSCALL_BASE_GAS_COST + 50 *
       ↪ STEP_GAS_COST;
86  +const GET_EXECUTION_INFO_GAS_COST = SYSCALL_BASE_GAS_COST + 10 *
       ↪ STEP_GAS_COST;

88  +const REPLACE_CLASS_GAS_COST = SYSCALL_BASE_GAS_COST + 50 *
       ↪ STEP_GAS_COST;
89  +const STORAGE_READ_GAS_COST = SYSCALL_BASE_GAS_COST + 50 *
       ↪ STEP_GAS_COST;
90  +const STORAGE_WRITE_GAS_COST = SYSCALL_BASE_GAS_COST + 50 *
       ↪ STEP_GAS_COST;
91  +const EMIT_EVENT_GAS_COST = SYSCALL_BASE_GAS_COST + 10 *
       ↪ STEP_GAS_COST;
92  +const SEND_MESSAGE_TO_L1_GAS_COST = SYSCALL_BASE_GAS_COST + 50 *
       ↪ STEP_GAS_COST;

94  +const SECP256K1_ADD_GAS_COST = SYSCALL_BASE_GAS_COST + 254 *
       ↪ STEP_GAS_COST + 29 *

96  +const SECP256K1_GET_POINT_FROM_X_GAS_COST = SYSCALL_BASE_GAS_COST +
       ↪  260 * STEP_GAS_COST + 30 *

98  +const SECP256K1_GET_XY_GAS_COST = SYSCALL_BASE_GAS_COST + 24 *
       ↪ STEP_GAS_COST + 9 *

100 +const SECP256K1_MUL_GAS_COST = SYSCALL_BASE_GAS_COST + 121810 *
       ↪ STEP_GAS_COST + 10739 *

102 +const SECP256K1_NEW_GAS_COST = SYSCALL_BASE_GAS_COST + 340 *
       ↪ STEP_GAS_COST + 36 *
```

CVF-22. INFO

- **Category** Bad datatype
- **Source** output.cairo

Recommendation This value should be a named constant.

Client Comment *This is defined only in the hint, we can an import of the constant but it will be pretty much the same.*

```
91 +max_page_size = 3800
```

CVF-23. FIXED

- **Category** Documentation
- **Source** os.cairo

Recommendation Consider writing explicitly which bound is asserted this way and why too big positive numbers are not a problem.

Client Comment *We'll add a comment that we're only storing block hashes >=10 blocks in the past.*

```
172 +let is_old_block_number_non_negative = is_nn(old_block_number);
```

CVF-24. INFO

- **Category** Unclear behavior
- **Source** signature.cairo

Description It is possible to pass the sign not in the parity bit but as a quadratic residue/non-residue in the starknet field. Then to assert the choice for y it is enough to verify that y/v is a square in the starknet field, with just a few multiplications.

```
243 +assert_nn((y.d0 + v) / 2);
```

CVF-25. FIXED

- **Category** Suboptimal
- **Source** keccak.cairo

Recommendation These functions are very similar and could be merged into one function with an additional argument "bigend".

Client Comment *Will unify.*

```
27 +func keccak_uint256s{range_check_ptr, bitwise_ptr: BitwiseBuiltin*,
   ↪  keccak_ptr: KeccakBuiltin*}(
```

```
42 +func keccak_uint256s_bigend{
```

CVF-26. FIXED

- **Category** Documentation
- **Source** keccak.cairo

Recommendation Consider explaining how field elements are converted to bitstrings.

Client Comment *Will document.*

```
55 +// Computes the keccak hash of multiple field elements.
56 +func keccak_felts{range_check_ptr, bitwise_ptr: BitwiseBuiltin*,
      ↪ keccak_ptr: KeccakBuiltin*}(
```

```
69 +// Computes the keccak hash of multiple field elements (big-endian)
      ↪ .
70 +// Note that both the output and the input are in big endian
      ↪ representation.
71 +func keccak_felts_bigend{range_check_ptr, bitwise_ptr:
      ↪ BitwiseBuiltin*, keccak_ptr: KeccakBuiltin*}(
```

CVF-27. FIXED

- **Category** Suboptimal
- **Source** keccak.cairo

Recommendation These functions are very similar and could be merged into one function with an additional argument “bigend”.

Client Comment *Will unify.*

```
56 +func keccak_felts{range_check_ptr, bitwise_ptr: BitwiseBuiltin*,
      ↪ keccak_ptr: KeccakBuiltin*}(
```

```
71 +func keccak_felts_bigend{range_check_ptr, bitwise_ptr:
      ↪ BitwiseBuiltin*, keccak_ptr: KeccakBuiltin*}(
```

CVF-28. FIXED

- **Category** Documentation
- **Source** keccak.cairo

Recommendation Consider describing the formula calculated by this function.

Client Comment *Will document.*

```
84 +// Converts a final state of the Keccak builtin to the hash output
      ↪ as `Uint256`.
```

CVF-29. INFO

- **Category** Suboptimal
- **Source** keccak.cairo

Recommendation These checks could be simplified as: `256**9 - 1 - output0_high` `256**7 - 1 - output1_low` `256**2 - 1 - output1_high`

Client Comment *Negligible (and the power operations are computed in compile time).*

```
99  +assert [range_check_ptr + 2] = output0_high - 256 ** 9 + 2 ** 128;
```

```
110 +assert [range_check_ptr + 6] = output1_low - 256 ** 7 + 2 ** 128;
111 +assert [range_check_ptr + 7] = output1_high - 256 ** 2 + 2 ** 128;
```

CVF-30. INFO

- **Category** Suboptimal
- **Source** keccak.cairo

Recommendation This could be simplified as: `256**n - 1 - inputs[i]`

Client Comment *Same as above.*

```
166 +assert [range_check_ptr + 1] = inputs[0] - 256 ** 8 + 2 ** 128;
```

```
168 +assert [range_check_ptr + 3] = inputs[1] - 256 ** 8 + 2 ** 128;
```

```
170 +assert [range_check_ptr + 5] = inputs[2] - 256 ** 8 + 2 ** 128;
```

```
176 +assert [range_check_ptr + 8] = low3 - 256 + 2 ** 128;
177 +assert [range_check_ptr + 9] = high3 - 256 ** 7 + 2 ** 128;
```

```
189 +assert [range_check_ptr + 11] = inputs[4] - 256 ** 8 + 2 ** 128;
```

```
191 +assert [range_check_ptr + 13] = inputs[5] - 256 ** 8 + 2 ** 128;
```

```
197 +assert [range_check_ptr + 16] = low6 - 256 ** 2 + 2 ** 128;
198 +assert [range_check_ptr + 17] = high6 - 256 ** 6 + 2 ** 128;
```

```
210 +assert [range_check_ptr + 19] = inputs[7] - 256 ** 8 + 2 ** 128;
```

```
212 +assert [range_check_ptr + 21] = inputs[8] - 256 ** 8 + 2 ** 128;
```

```
218 +assert [range_check_ptr + 24] = low9 - 256 ** 3 + 2 ** 128;
219 +assert [range_check_ptr + 25] = high9 - 256 ** 5 + 2 ** 128;
```

```
231 +assert [range_check_ptr + 27] = inputs[10] - 256 ** 8 + 2 ** 128;
```

```
233 +assert [range_check_ptr + 29] = inputs[11] - 256 ** 8 + 2 ** 128;
```

```
239 +assert [range_check_ptr + 32] = low12 - 256 ** 4 + 2 ** 128;
240 +assert [range_check_ptr + 33] = high12 - 256 ** 4 + 2 ** 128;
```

```
252 +assert [range_check_ptr + 35] = inputs[13] - 256 ** 8 + 2 ** 128;
```

```
254 +assert [range_check_ptr + 37] = inputs[14] - 256 ** 8 + 2 ** 128;
```

```
260 +assert [range_check_ptr + 40] = low15 - 256 ** 5 + 2 ** 128;
261 +assert [range_check_ptr + 41] = high15 - 256 ** 3 + 2 ** 128;
```

```
273 +assert [range_check_ptr + 43] = inputs[16] - 256 ** 8 + 2 ** 128;
```

CVF-31. INFO

- **Category** Suboptimal
- **Source** keccak.cairo

Recommendation This could be simplified as: MAX_VALUE - 1 - x

Client Comment *Same as above.*

```
307 +assert [range_check_ptr + 2] = n_bytes_left - BYTES_IN_WORD + 2 **
    ↪ 128;
308 +assert [range_check_ptr + 3] = n_words_to_copy -
    ↪ KECCAK_FULL_RATE_IN_WORDS + 2 ** 128;
```

CVF-32. FIXED

- **Category** Readability
- **Source** keccak.cairo

Recommendation Consider providing the keccak padding scheme in comments.

Client Comment *Will document.*

```
322 +let first_one = _pow256(n_bytes_left);
323 +// The beginning of the padding with the last bytes of the input
    ↪ and the first 1.
324 +let input_word_with_initial_padding = input_word + first_one;
325 +
326 +if (padding_len == 1) {
327 +    assert dst[0] = 2 ** 63 + input_word_with_initial_padding;
328 +} else {
329 +    // Padding of more than 1 word.
330 +    assert dst[0] = input_word_with_initial_padding;
331 +    memset(dst=dst + 1, value=0, n=padding_len - 2);
332 +    assert dst[padding_len - 1] = 2 ** 63;
333 +}
```

CVF-33. FIXED

- **Category** Procedural
- **Source** hash_state_poseidon.cairo

Description This import is not used.

Recommendation Consider removing it.

Client Comment *Will remove.*

```
4 +poseidon_hash_single,
```

CVF-34. INFO

- **Category** Procedural
- **Source** patricia_with_poseidon.cairo

Description We didn't review this function.

```
5 +patricia_update_using_update_constants as
     ↪ patricia_update_using_update_constants_with_sponge,
```

CVF-35. INFO

- **Category** Unclear behavior
- **Source** patricia.cairo

Description These comments look like instructions for some kind of macroprocessor, but its unclear what code they are expanded to. Without this knowledge, it is impossible to tell whether the code is correct or not.

Client Comment *These are used only here to avoid duplicating patricia.cairo for both pedersen/poseidon, we'll either document and add the generating code to the scope (see patricia_gen_test.py) or remove this mechanism alltogether and duplicate.*

```
20 +// ADDITIONAL_IMPORTS_MACRO()
```

```
46 +    // PREPARE_ADDITIONAL_HASH_INPUTS_MACRO(hash_ptr)
```

```
250 +                // PREPARE_ADDITIONAL_HASH_INPUTS_MACRO(hash_ptr)
```

```
325 +    // PREPARE_ADDITIONAL_HASH_INPUTS_MACRO(current_hash)
```

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting