Security Review Report
NM-0061: Cartdrige - SHA-256 Implementation

NETHERMIND

(Oct 24, 2022)

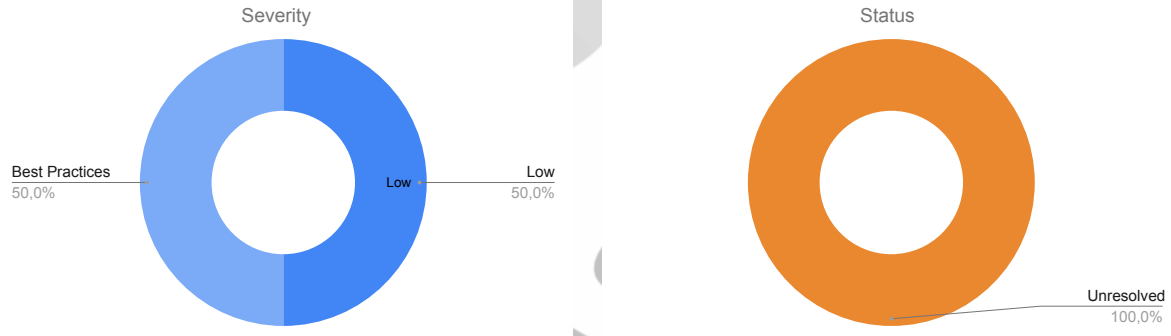
Contents

1 Executive Summary

This document presents the security review performed by Nethermind on the SHA-256 implementation written in the Cairo language by the **Cartridge** team. This project implements the `SHA-256` [1] algorithm without imposing limits on the input length (the `SHA-256` can operate with inputs up to $(2^{64}-1)$ bits. The code is an extension of the StarkWare implementation. The `SHA-256` is a part of the `SHA-2` family of algorithms, where `SHA` stands for `Secure Hash Algorithm`. Published in 2001, it was a joint effort between the National Security Agency (NSA) and the National Institute of Standards and Technology (NIST) to introduce a successor to the `SHA-1` family [2, 3], which was slowly losing strength against brute force attacks.

During this audit the source code was analyzed in detail, including: **a) line-by-line inspection of the codebase; b) manual execution of the algorithm checking for all branches; c) white-box testing; d) black-box testing**. For a reference on techniques for performing white-box and black-box tests, please check [4]. **During the black-box tests, we have evaluated more than 28,000 inputs**, ranging from 0 bits up to 2^{20} bits. Due to hardware limitations, we could not execute inputs above this threshold. It is worth mentioning that the `SHA-256` algorithm accepts inputs of length up to $(2^{64}-1)$ bits, and this limit is not enforced in the code.

The codebase is composed of 522 lines of Cairo code having 64 lines of comments (comments ratio of 12.3%). **During the initial audit, we point out 2 potential issues**. The distribution of issues is summarized in the figure below. **This document is organized as follows.** Section 2 presents the files in the scope of this audit. Section 3 summarizes the findings in a table. Section 4 discusses the risk rating methodology adopted for this audit. Section 5 details each finding. Section 5 details the white-box and black-box tests performed. Section 7 discusses the documentation provided for this audit. Section 8 presents the output of the automated test suite. Section 9 concludes the audit report.



(a) (b)

Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (0), **Best Practices** (1).
Distribution of status: Unresolved (2)

Summary of the Audit

Audit Type	Security Review
Initial Report	Oct. 24, 2022
Response from Client	-
Final Report	-
Methods	Manual Review, White-box testing, Black-box testing, Manual logic flow analysis
Repository	Cartridge
Commit Hash (Initial Audit)	4b364b1504e670646da4c06efe0caecdc1e6689d
Documentation	README.md
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Contracts

	Contract	Lines of Code	Lines of Comments	Comments Ratio	Blank Lines	Total Lines
1	src/packed_sha256.cairo	247	18	7.3%	22	287
2	src/sha256.cairo	275	46	16.7%	58	379
	Total	**522**	**64**	**12.3%**	**80**	**666**

3 Summary of Issues

	Finding	Severity	Update
1	Wrong use of `is_le(...)`	Low	Unresolved
2	Input is not ensured to be less than (2^64 - 1) bits	Best Practices	Unresolved

4 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to: Data/state integrity, loss of availability, financial loss, reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

5 Detailing the Audit

This section describes each part of the overall audit process. The audit team was divided into groups responsible for:

a) Line-by-line inspection of the source code;

b) Manual execution of the algorithm on pen and paper covering all possible code branches;

c) White box testing targeting particular logic

d) Black box testing

These groups worked cooperatively, sharing knowledge to improve the overall understanding of the code and investigate potential attack vectors. These groups were supported by the Nethermind cryptography team.

5.1 White-box tests

White box testing involves testing an application with detailed inside information of its source code, architecture and configuration. It can expose issues such as security vulnerabilities, broken paths or data flow issues, which black box testing cannot test comprehensively. **The white-box tests were divided into specific goals:**

a) Validate if every line of code is tested at least once;

b) Improve code auditing by scanning the functions `sha256(...)` and `finalize_sha256(...)` and their dependencies for potential runtime security issues caused by the `hints`.

5.1.1 Methodology

Our methodology consists in two stages: **i) careful inspection of the code**, line-by-line, annotating parts of the code that require deeper investigation; **ii) manual code inspection combined with dynamic code analysis** during the auditing process. Dynamic analysis is a debugging technique to test and evaluate a program while it is running. The code must be instrumented to extract the test cases execution traces for monitoring the dynamic interaction. We generate a trace file during the execution of the functions `sha256(...)` and `finalize_sha256(...)`. This file presents the order of the code that is executed during the test cases, such as called functions, variables, and executed branches for a given input. Then, for each test case, we reinspect the code following the captured traces.

5.1.2 Statement and Branch Coverage

We applied two **White Box Test design techniques**: **i) statement coverage** that is used to verify if every line of code has been executed at least once; and, **ii) branch coverage** that is used to ensure that each decision condition from every branch has been tested at least once.

```
sha256
  |--- sha256_inner
         if (is_last_block == 1)
              |--- _sha256_input(...)
              |--- _sha256_chunk(...)
         if (is_remainder_block == 1)
              |--- _sha256_input(...)
              |--- _sha256_chunk(...)
              |--- sha256_inner(...)
         else
              |--- _sha256_input(...)
              |--- _sha256_chunk(...)
              |--- sha256_inner(...)
```

```
_sha256_input
   if (full_word != 0)
        |--- _sha256_input(...)
   if (n_words == 0)
        |--- return()
   if (n_bytes == 0 and pad_chunk == 1)
        |--- return()
   if (n_bytes == 0)
        |--- return()
 return ()
```

Figure 1: Branches in `sha256_inner` and `_sha256_input`

Branch Coverage: We executed 8 (eight) test cases of arbitrary length on the instrumented code to validate all branches and to ensure that no branch led to any unexpected behavior. Fig. 1 describes the existent branches in functions `sha256_inner` and `_sha256_input`. **Results:** Every branch of the code was executed at least once. The following table lists the test cases used for white-box tests and the **branch coverage measure**. As we can note, the coverage for shorter messages is lower than the coverage for longer messages. Each branch is executed based on the message length, thus, this raises no concerns at all.

Table 2: Branch Coverage obtained by analyzing execution traces

Test Case	Branch Coverage
88 bits (hello_world)	37%
480 bits (multi_chunks)	37%
0 bits	25%
24 bits	25%
448 bits	37%
504 bits	37%
512 bits	37%
1,024 bits	37%
896 bits	37%
1,480 bits (client_data)	37%

Statement Coverage: We execute the same 8 (eight) test cases described in the table above on the instrumented code to ensure that there is no dead code, unused statements, or missing statements. **Result:** All lines of code were executed at least once.

5.1.3 Assessment of `Hints` in Functions

Figs. 2 and 3 describe the functions `_sha256_chunk(...)` and `finalize_sha256(...)`, respectively. The function `_sha256_chunk(...)` computes the `SHA-256` hash of arbitrary length messages inside the `hint` by using the Python `cairo_sha256`. This function is called by the `sha256_inner(...)` (see branches for the function in Fig. 1). For security reasons, the function `finalize_sha256(...)` is responsible to verify that the result of `sha256(...)` is valid. However, the function `finalize_sha256(...)` also includes `hints`, as described in Fig. 3.

```
func _sha256_chunk{range_check_ptr, message: felt*, state: felt*, output: felt*}() {
    %{
        from starkware.cairo.common.cairo_sha256.sha256_utils import (
            compute_message_schedule, sha2_compress_function)

        _sha256_input_chunk_size_felts = int(ids.SHA256_INPUT_CHUNK_SIZE_FELTS)
        assert 0 <= _sha256_input_chunk_size_felts < 100
        _sha256_state_size_felts = int(ids.SHA256_STATE_SIZE_FELTS)
        assert 0 <= _sha256_state_size_felts < 100
        w = compute_message_schedule(memory.get_range(
            ids.message, _sha256_input_chunk_size_felts))
        new_state = sha2_compress_function(memory.get_range(ids.state, _sha256_state_size_felts), w)
        segments.write_arg(ids.output, new_state)
    %}
    return ();
}
```

Figure 2: Computes the `sha256` hash of the input chunk

Now, we combine code manual inspection and dynamic code analysis to verify whether the functions are vulnerable to malicious provers, since `hints` can be manipulated by provers. The function `finalize_sha256(...)` calls `_finalize_sha256_inner(...)` to compute the `SHA256` hash and verify if the output of the function `sha256(...)` is valid. **Conclusion:** Since Cairo memory is write-once, we can only write one value to each memory cell. The message used by `_finalize_sha256_inner(...)` to compute the hash has been written before in Cairo code. There were no issues detected in this case where a malicious prover could change the message to manipulate the validation process implemented by `finalize_sha256(...)`.

```
func finalize_sha256{range_check_ptr, bitwise_ptr: BitwiseBuiltin*}(
    sha256_ptr_start: felt*, sha256_ptr_end: felt*
) {
    alloc_locals;
    // ...
    %{
        from starkware.cairo.common.cairo_sha256.sha256_utils import (
            IV, compute_message_schedule, sha2_compress_function)

        _block_size = int(ids.BLOCK_SIZE)#7
        assert 0 <= _block_size < 20
        _sha256_input_chunk_size_felts = int(ids.SHA256_INPUT_CHUNK_SIZE_FELTS)#16
        assert 0 <= _sha256_input_chunk_size_felts < 100

        message = [0] * _sha256_input_chunk_size_felts
        w = compute_message_schedule(message)
        output = sha2_compress_function(IV, w)
        padding = (message + IV + output) * (_block_size - 1)
        segments.write_arg(ids.sha256_ptr_end, padding)
    %}

    let (local q, r) = unsigned_div_rem(n + BLOCK_SIZE - 1, BLOCK_SIZE);
    _finalize_sha256_inner(sha256_ptr_start, n=q, round_constants=round_constants);
    return ();
}
```

Figure 3: `finalize_sha256` verifies that the results of `sha256(...)` are valid.

In this work, we combine black-box testing with white-box testing. By combining black-box and white-box testing, testers can achieve a comprehensive ”inside out” inspection of a software and increase the coverage of quality aspects and security issues. The black-box tests are discussed in the following subsection.

5.2 Black-box tests

Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. Test cases are generally derived from external descriptions of the software, including specifications, requirements and design parameters. Although the tests used are primarily functional in nature, non-functional tests may also be used. The test designer selects both valid and invalid inputs and determines the correct output, often with the help of a test oracle or a previous result that is known to be correct, without any knowledge of the test object’s internal structure. **The black-box tests were divided into specific goals:**

a) Validate if hashes of small messages are properly generated;

b) Increment the message length gradually until reaching very large messages;

c) Evaluate the time required the compute the hash of a given message.

5.2.1 Methodology adopted in the black-box tests

The generation of test cases has been automated with Python programs, which can be made available to the client at the end of this audit. We generated two families of programs:

a) Programs for generating test cases for Protostar;

b) Programs for calling the contract inside the Python program.

We have evaluated the correctness of the hash using messages generated from 0 to 2^{20} bits. Each message was randomly generated using the current Unix timestamp as the seed to determine the input and expected output.

5.2.2 Characterizing the sample

This section characterizes the number of random messages generated according to the number of bits employed in its construction. The results are shown in Fig. 4, where the x-axis indicates the number of random messages generated, while the y-axis indicates the number of bits used for composing the message. We note that 1,412 messages have been generated using between 0 and 512 bits, while 117 messages have been generated using 2^{20} bits. **All tests have passed.**

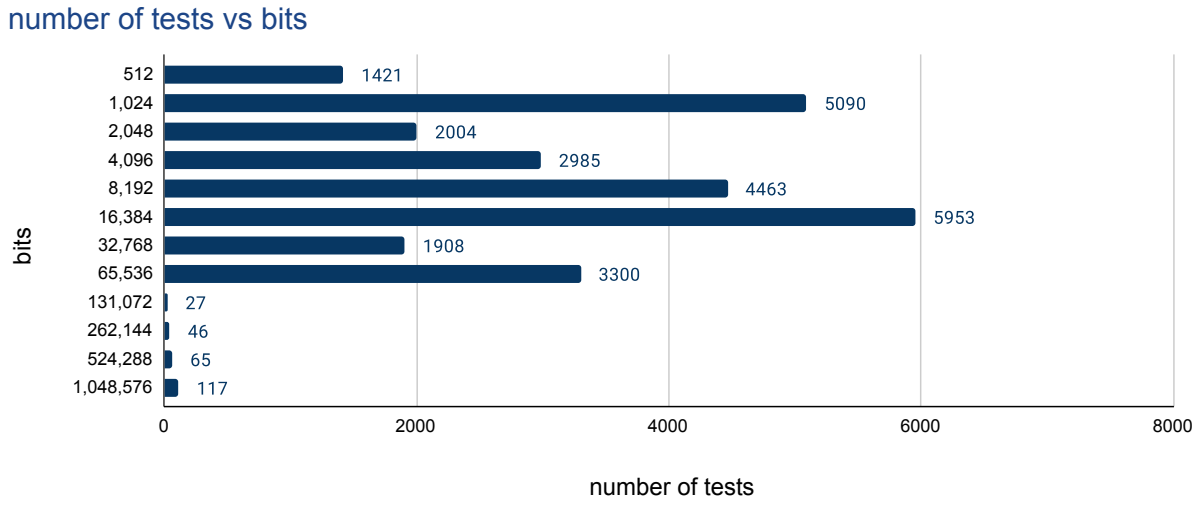
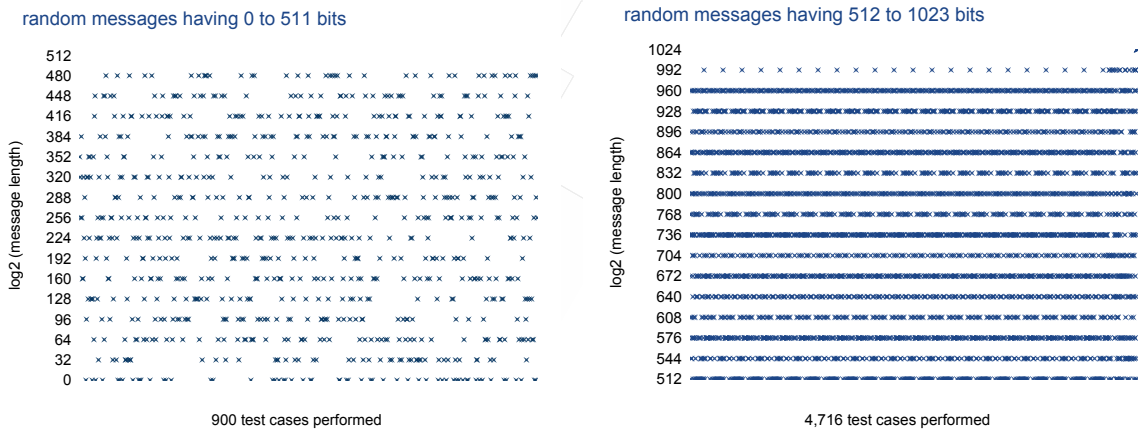


Figure 4: Histogram of messages length: the plot presents the number of random messages generated according to the number of bits employed. The x-axis indicates the number of random messages generated, while the y-axis indicates the number of bits used for composing the message.

Most of the tests are concentrated in messages having 0 to 65,536 bits because this seems to be a typical interval where most of the messages will be concentrated on. Larger messages require a high execution time, which is unfeasible for mass testing.

5.2.3 Testing the SHA-256 for random messages generated from 0 to 1023 bits

This experiment is broken into two stages. In the first stage, we generate 900 random messages consisting of between 0 and 511 bits using a Python program to create test cases for Protostar. The results are presented in Fig. 5(a). In the second stage, we generate 4,716 random messages consisting of between 512 and 1023 bits. The results are presented in Fig. 5(b). **All tests have passed.**



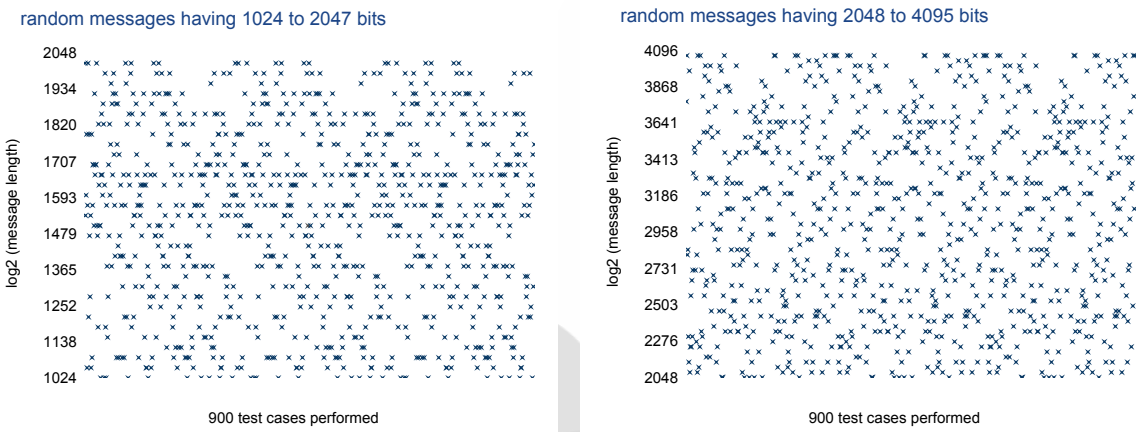
(a) 0-511 bits

(b) 512-1023 bits

Figure 5: Distribution of the length of the random messages tested. Fig. (a) considers messages having 0-511 bits, while Fig. (b) considers messages generated from 512-1023 bits.

5.2.4 Testing the SHA-256 for random messages generated from 1,024 to 4,095 bits

Again, this experiment is broken into two stages. In the first stage, we generate 900 random messages consisting of between 1,024 and 2,047 bits. In the second stage, we generate 900 random messages using between 2,048 and 4,095 bits. The results are presented in Figs. 6(a) and 6(b). **All tests have passed.**

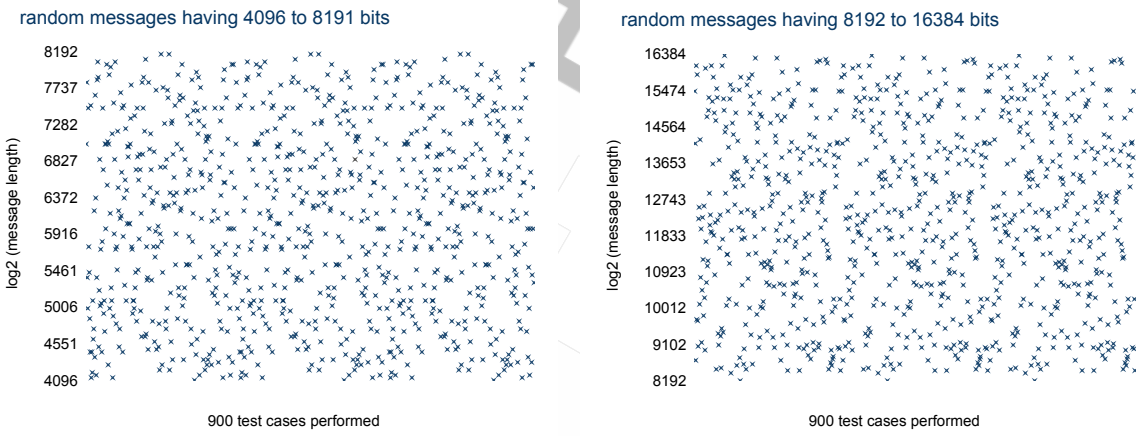


(a) 1,024-2,047 bits (b) 2,048-4,095 bits

Figure 6: Distribution of the length of the random messages tested. Fig. (a) considers messages having between 1,024 and 2,047 bits, while Fig. (b) considers messages generated from between 2,048 and 4,095 bits.

5.2.5 Testing the SHA-256 for random messages generated from between 4,096 and 16,384 bits

Again, this experiment is broken into two stages. In the first stage, we generate 900 random messages having between 4,096 and 8,191 bits. In the second stage, we generate 900 random messages using between 8,192 and 16,384 bits. The results are presented in Figs. 6(a) and 6(b). **All tests have passed.**



(a) 4,096-8,191 bits (b) 8,192-16,384 bits

Figure 7: Distribution of the length of the random messages tested. Fig. (a) considers messages having between 4,096 and 8,191 bits, while Fig. (b) considers messages generated from between 8,192 and 16,384 bits.

5.2.6 Testing long messages (up to 2^{20} bits)

When the test cases are predefined and limited to smaller bits, the tests can run quickly. But for tests, with higher bits, the test consumes a lot of time and memory. To overcome this situation, we used the `Python hashlib` module to get the `SHA-256` and used the Python random library for data/message generation. We used the `Cairo Function Runner` to run the tests. The time taken to complete each test was also measured. By adopting this strategy, we could validate test cases up to 2^{20} bits, and create this experiment. **This experiment shows the distribution of the tests according to the number of bits used**. It combines the data generated from both `Python` test programs. The sample is composed of 27,382 test cases, as shown in Fig. 8. **All tests have passed.**

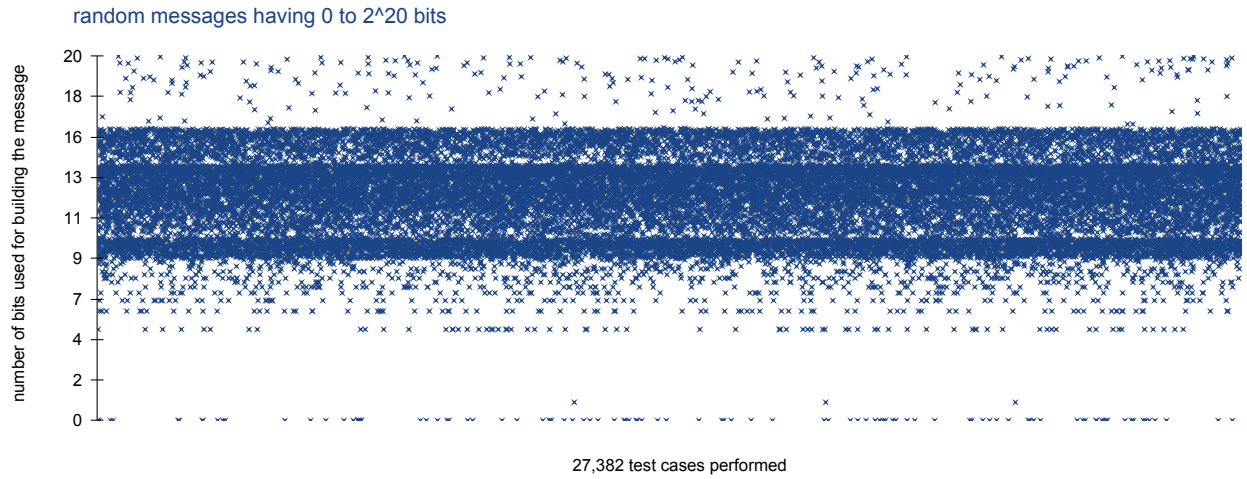


Figure 8: Distribution of the 27,382 test cases generated for messages ranging from 0 up to 2^{20} bits.

5.2.7 Assessment of the time for running the tests

The time for running each test instance is presented in Fig. 9. The x-axis indicates the message length (bits), while the y-axis indicates the time (seconds) for running the test case. We notice that small test cases run very fast. However, long messages having up to 2^{20} bits can take more than one hour of computing time.

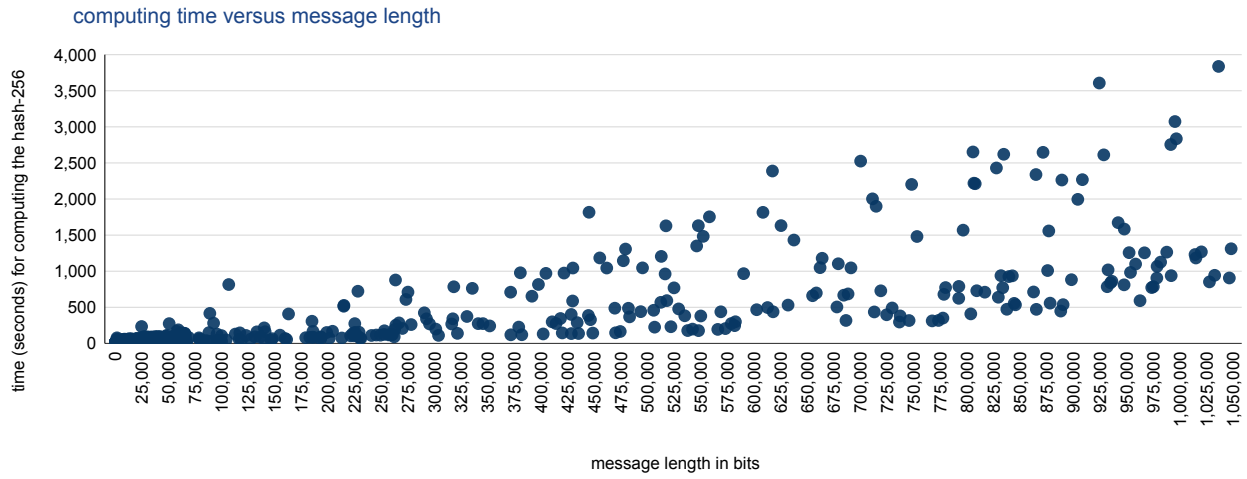


Figure 9: Assessment of the time required for running the tests. The x-axis indicates the message length (bits), while the y-axis indicates the time (seconds) for running the test case.

5.2.8 Final Remarks

The white-box inspection of the code did not reveal any issue. Similarly, the black-box tests have not indicated any issue after the inspection of 27,382 test cases from sizes ranging from 0 up to 2^{20} bits.

6 Issues

6.1 src/sha256.cairo

6.1.1 [Low] Wrong use of is_le(...)

File(s): src/sha256.cairo

Description: The function `is_le(...)` is used for comparing `felts`. This function in fact checks if `(b - a)` is within the range $[0, 2^{128})$ and should only be used for comparing members of the `Uint256` struct. For comparing unsigned `felts` of arbitrary values in the range `[0, P)` use the function `is_le_felt(...)`.

Recommendation(s): Replacing the function `is_le(...)` to `is_le_felt(...)`, which assumes arguments in the range `[0, P)`.

Status: Unresolved

Update from the client:

6.1.2 [Best Practices] Input is not ensured to be less than (2^64 - 1) bits

File(s): src/sha256.cairo

Description: The `SHA-256` is defined for a maximum input length of (2^{64}-1) bits. The code does not validate the input length. Although in real scenarios such input won't be processed due to StarkNet steps limitations, it is important to be in conformance with the `SHA-256` specification.

Recommendation(s): Ensure that the input length is below (2^{64}-1) bits.

Status: Unresolved

Update from the client:

7 Documentation Evaluation

Technical documentation is created to explain what the software product does. This way, developers and stakeholders can easily follow the purpose and the underlying functionality of each file/function/line. Documentation can come not only in the form of a `README.md` but also using code as documentation (to write clear code), diagrams, websites, research papers, videos and external documentation. Besides being a good programming practice, proper technical documentation improves the efficiency of audits. Less time can be spent understanding the protocol and more time can be put towards auditing which improves the efficiency and overall output of the audit. **Since the `SHA-256` algorithm is widely known**, the only documentation provided for the audit was the `README.md` file. It describes how to use the `sha256` function, explains the need for zero byte padding and shows how to run the automated test suite.

The README.md states that the code *"computes SHA256 of 'input'. Inputs of arbitrary length are supported"*, which is not true. The specification of the `SHA-256` algorithm limits the input up to $2^{64} - 1$ bits. Also notice that StarkNet has a limit of computation allowed in order to prevent halting the Blockchain.

8 Test Suite Evaluation

In this section we present the output of the compilation process and test suite provided by the client.

8.1 Contracts Compilation

```
cairo-sha256-main % cairo-compile src/sha256.cairo --output compiled/sha256_compiled.json && cairo-compile
↪  src/packed_sha256.cairo --output compiled/packed_sha256_compiled.json
```

8.2 Tests Output

The test suite provided by the client has 10 test cases, and can be found at test_sha256.cairo. The output of the tests is presented below.

```
dev@dev:~/NM-0061/cairo-sha256$ protostar test
15:43:49 [INFO] Collected 1 suite, and 10 test cases (0.498 s)
[PASS] tests/test_sha256.cairo test_sha256_multichunks (time=3.48s, steps=12620, memory_holes=218)
       range_check_builtin=469 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_0bits (time=1.12s, steps=12176, memory_holes=86)
       range_check_builtin=458 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_504bits (time=1.19s, steps=12653, memory_holes=217)
       range_check_builtin=471 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_1024bits (time=0.80s, steps=13059, memory_holes=361)
       range_check_builtin=480 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_512bits (time=1.15s, steps=12619, memory_holes=222)
       range_check_builtin=469 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_hello_world (time=0.75s, steps=12240, memory_holes=78)
       range_check_builtin=460 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_24bits (time=1.13s, steps=12220, memory_holes=76)
       range_check_builtin=460 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_448bits (time=1.19s, steps=12611, memory_holes=217)
       range_check_builtin=469 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_896bits (time=0.80s, steps=12760, memory_holes=225)
       range_check_builtin=469 bitwise_builtin=1560
[PASS] tests/test_sha256.cairo test_sha256_client_data (time=1.23s, steps=13564, memory_holes=489)
       range_check_builtin=494 bitwise_builtin=1560

15:44:15 [INFO] Test suites: 1 passed, 1 total
15:44:15 [INFO] Tests:       10 passed, 10 total
15:44:15 [INFO] Seed:        2038124827
15:44:16 [INFO] Execution time: 29.49 s
```

9 About Nethermind

Founded in 2017 by a small team of world-class technologists, Nethermind builds Ethereum solutions for developers and enterprises. Boosted by a grant from the Ethereum Foundation in August 2018, our team has worked tirelessly to deliver the fastest Ethereum client in the market. Our flagship Ethereum client is all about performance and flexibility. Built on .NET core, a widespread, enterprise-friendly platform, Nethermind makes integration with existing infrastructures simple, without losing sight of stability, reliability, data integrity, and security

Nethermind is made up of several engineering teams across various disciplines, all collaborating to realize the Ethereum roadmap, by conducting research and building high-quality tools. Teams focus on specific areas of the Ethereum problem space. Each consists of specialists and experienced developers working alongside interns, learning the ropes in the Nethermind Internship Program.

Our mission is to gather passionate talent from around the world, and to tackle some of the blockchain's most complex problems. Nethermind provides software solutions and services for developers and enterprises building the Ethereum ecosystem. We offer security reviews to projects built on EVM compatible chains and StarkNet. We have expertise in multiple areas of the Ethereum ecosystem, including protocol design, smart contracts (written in Solidity and Cairo), MEV, etc. We develop some of the most used tools on Starknet and one of the most used Ethereum clients. Learn more about us at https://nethermind.io.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

References

[1] Wouter Penard and Tim van Werkhoven. On the secure hash algorithm family. *Cryptography in context*, pages 1–18, 2008.

[2] D Eastlake 3rd and Paul Jones. Us secure hash algorithm 1 (sha1). Technical report, 2001.

[3] D Eastlake 3rd and Paul Jones. Rfc3174: Us secure hash algorithm 1 (sha1), 2001.

[4] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.