Data Fragmentation and Reconstruction Challenge

Important Note: While the scenario and task descriptions below use Python syntax for illustration, **please solve this challenge in the language specified by the recruiter.**

Scenario: In a near-future cyberpunk setting, safeguarding information is paramount. Data is frequently broken down into fragments and dispersed across multiple storage units to enhance security and resilience against data breaches. To ensure the integrity of this fragmented data, each piece is stored along with a hash value calculated using a proprietary hashing algorithm. When the data needs to be reconstructed, the integrity of each fragment must be verified against its hash before the reconstruction process can proceed.

Task: Your challenge is to implement a function, `reconstruct_data`, that reassembles the original data from its fragments. This function must verify the integrity of each fragment using its corresponding hash value prior to the reconstruction. You are also tasked with developing a simplistic yet effective hashing algorithm, `simple_hash`, which will be used for generating and verifying the hash values of the data fragments.

Input:

- A dictionary named `fragments`, where:
  - Keys are integers representing the sequence of the fragments.
  - Values are dictionaries containing two keys: `data` (a string representing the data fragment) and `hash` (a string representing the hash value of the fragment).
- The `simple_hash` function will take a single string argument and return a fixed-length (30 characters) hash value as a string.

Expected Output:

- The `reconstruct_data` function should return a string representing the reconstructed data if all fragments are intact and in the correct order.
- If the integrity verification of any fragment fails (i.e., the recalculated hash of a fragment does not match the stored hash), the function should return an error message: `"Error: Data integrity verification failed."`

Constraints:

- The `simple_hash` function should not use any external cryptographic libraries but should still be designed to minimize collisions.

- The hash function should produce a fixed-length output (30 characters) to ensure consistency across different data fragments.
- Consider the efficiency and complexity of your implementation, as it may need to handle a large number of data fragments.

Example:

```
# your implementation of the simple_hash function
# your implementation of the reconstruct_data function

fragments = {
    1: {'data': 'Hello', 'hash': simple_hash('Hello')},
    2: {'data': 'World', 'hash': simple_hash('World')},
    3: {'data': '!', 'hash': simple_hash('!')}
}

original_data = reconstruct_data(fragments)
print(fragments)
print(original_data)
```

Evaluation Criteria:

- Correctness: The `reconstruct_data` function accurately reconstructs the original data when all fragment hashes verify correctly.
- Hash Function Design: The `simple_hash` function exhibits a good balance between simplicity and collision resistance within the constraints of the challenge.
- Code Quality: The code is well-structured, readable, and efficiently handles the reconstruction and verification process.
- Edge Case Handling: The solution adequately addresses potential edge cases, such as out-of-order fragments or missing data.