Project: CryptoCore - Technical Requirements Document (Sprint 4)

Sprint Goal: Add cryptographic hash functions for verifying data integrity.

1. Project Structure & Repository Hygiene

The codebase must be extended to support hash functionality while maintaining existing structure.

ID	Requirement Description	Priority
STR-1	All requirements from previous Sprints (STR-1 to STR-4) **must** still be met.	Must
STR-2	New source files for hash implementations **must** be created in a logical directory structure. - **Suggested Path:** `src/hash/` containing `sha256.py`/`sha256.c`, `sha3_256.py`/`sha3_256.c`, etc.	Must
STR-3	The `README.md` file **must** be updated to include: - Documentation for the new `dgst` command and its options. - Examples of computing hashes for files. - Information about the implemented hash algorithms and their security properties.	Must
STR-4	The build system **must** be updated to include any new source files for hash implementations.	Must

2. Command-Line Interface (CLI) Parser

A new subcommand must be implemented for hash operations, separate from the encryption/decryption functionality.

ID	Requirement Description	Priority
CLI-1	The tool **must** support a new subcommand `dgst` for computing message digests.	Must
CLI-2	The `dgst` subcommand **must** accept the following arguments: - `--algorithm ALGORITHM` : **Must** accept at least `sha256` and one other algorithm (e.g., `sha3-256`, `blake2`). - `--input INPUT_FILE` : **Must** accept a filesystem path to the input file to be hashed.	Must
CLI-3	The `dgst` command **must not** require or accept encryption-specific arguments (`--key`, `--mode`, `--encrypt`/`--decrypt`, `--iv`).	Must
CLI-4	The output **must** be printed to `stdout` in the format: `HASH_VALUE INPUT_FILE_PATH` (matching the standard `*sum` tool format).	Must
CLI-5	The tool **should** support an optional `--output FILE` flag to write the hash to a file instead of `stdout`.	Should

Example Invocations:

```
# Basic hash computation
$ cryptocore dgst --algorithm sha256 --input document.pdf
> 5d5b09f6dcb2d53a5fffc60c4ac0d55fb052072fa2fe5d95f011b5d5d5b0b0b5  document.pdf

# Hash with output to file
$ cryptocore dgst --algorithm sha3-256 --input backup.tar --output backup.sha3
```

3. Hash Function Implementation

This sprint involves significant cryptographic implementation work for the hash functions.

ID	Requirement Description	Priority
HASH-1	**SHA-256 must** be implemented from scratch by the student. - The implementation **must** follow the SHA-256 specification (NIST FIPS 180-4). - It **must** process input in 512-bit blocks using the Merkle-Damgård construction. - It **must** correctly implement the padding scheme (append bit '1', then '0's, then 64-bit message length). - It **must** implement all SHA-256 constants (fractional parts of cube roots of primes, fractional parts of square roots of primes) and round functions.	Must
HASH-2	A second hash algorithm **must** be implemented. The student **should** choose one of: - **SHA3-256** (from scratch, following NIST FIPS 202, using Keccak sponge construction) - **BLAKE2b** (from scratch, following RFC 7693)	Must

ID	Requirement Description	Priority
HASH-3	Alternatively, if the course focus is on understanding hash functions rather than low-level implementation, the second algorithm **may** be implemented using a vetted library (e.g., Python's `hashlib`, OpenSSL's `EVP` digest functions).	Could
HASH-4	All hash implementations **must** support input of arbitrary length (including empty files and very large files).	Must
HASH-5	The implementations **must** process files in chunks to maintain constant memory usage regardless of input size.	Must
HASH-6	The hash functions **must** produce output as lowercase hexadecimal strings.	Must

Expected SHA-256 Implementation Structure:

```
# Python: src/hash/sha256.py
class SHA256:
    def __init__(self):
        # Initialize hash values (first 32 bits of fractional parts of square roots of first 8 primes)
        self.h = [0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
                  0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19]
        # Initialize round constants (first 32 bits of fractional parts of cube roots of first 64 primes)
        self.k = [...]

    def padding(self, message):
        # Implement SHA-256 padding
        pass

    def process_block(self, block):
        # Process one 512-bit block
        pass

    def update(self, message):
        # Process message in blocks
        pass

    def digest(self):
        # Return final hash
        pass
```

4. File I/O for Hashing

The hash functionality must handle files efficiently and correctly.

ID	Requirement Description	Priority
IO-1	The tool **must** read the input file in binary mode (`'rb'` in Python).	Must
IO-2	The implementation **must** process files in chunks (e.g., 4096 or 8192 bytes) to handle files larger than available memory.	Must
IO-3	If the `--output` flag is provided, the tool **must** write the hash output in the same format as would be printed to `stdout`.	Must
IO-4	The tool **must** handle file errors gracefully (e.g., missing input file) with clear error messages.	Must

5. Testing & Verification

Comprehensive testing must ensure correctness and interoperability with standard tools.

ID	Requirement Description	Priority
TEST-1	**Known-Answer Tests:** The implementations **must** pass all NIST-provided test vectors for each implemented algorithm. - Test vectors are available from NIST websites for SHA-256 and SHA3-256.	Must
TEST-2	**Empty Input Test:** Hashing an empty file **must** produce the correct hash (e.g., SHA-256 of empty string: `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`).	Must
TEST-3	**Interoperability Test:** For every implemented algorithm, the tool's output **must** match the corresponding system command: - `sha256sum <file>` for SHA-256	Must

ID	Requirement Description	Priority
	- `sha3sum -a 256 <file>` for SHA3-256	
	- `b2sum -l 256 <file>` for BLAKE2b-256	
TEST-4	**Large File Test:** The implementation **must** correctly hash files larger than 1GB (verifying chunk processing works correctly).	Must
TEST-5	**Avalanche Effect Test:** A test **should** be created that verifies changing one bit in the input produces a completely different hash.	Should
TEST-6	**Performance Test:** The student **should** measure and document the performance of their implementation compared to the system tool for various file sizes.	Could

Example Test Commands:

```
# Test with known vectors
$ echo -n "abc" | cryptocore dgst --algorithm sha256 --input -
> ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad  -

# Interoperability test
$ cryptocore dgst --algorithm sha256 --input large_file.iso > my_hash.txt
$ sha256sum large_file.iso > system_hash.txt
$ diff my_hash.txt system_hash.txt  # Should show no differences

# Test with NIST test vectors (example for SHA-256)
$ echo -n "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq" | cryptocore dgst --algorithm sha256 --input -
> 248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1  -
```

Example Test Script for Avalanche Effect:

```
# tests/test_hash_avalanche.py
from src.hash.sha256 import SHA256

def test_avalanche_effect():
    """Test that changing one bit produces completely different hash"""
    original_data = b"Hello, world!"
    modified_data = b"Hello, world?"  # Changed last character

    sha256 = SHA256()
    hash1 = sha256.hash(original_data)
    sha256 = SHA256()  # Reset
    hash2 = sha256.hash(modified_data)

    # Convert to binary and count differing bits
    bin1 = bin(int(hash1, 16))[2:].zfill(256)
    bin2 = bin(int(hash2, 16))[2:].zfill(256)

    diff_count = sum(bit1 != bit2 for bit1, bit2 in zip(bin1, bin2))

    print(f"Bits changed: {diff_count}/256")
    # Avalanche effect: should be ~128 bits changed (50%)
    assert 100 < diff_count < 156, f"Avalanche effect weak: only {diff_count} bits changed"
```