Project: CryptoCore - Technical Requirements Document (Sprint 5)

Sprint Goal: Implement functions to provide data authenticity and integrity.

1. Project Structure & Repository Hygiene

The codebase must be extended to support MAC 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 MAC implementations **must** be created in a logical directory structure. - **Suggested Path:** `src/mac/` containing `hmac.py/hmac.c`, `cmac.py/cmac.c`	Must
STR-3	The `README.md` file **must** be updated to include: - Documentation for the new `--hmac` and `--key` options for the `dgst` command. - Examples of generating and verifying HMACs. - Explanation of HMAC construction and its security properties. - If implemented, documentation for AES-CMAC.	Must

2. Command-Line Interface (CLI) Parser

The existing `dgst` command must be extended to support MAC operations with keys.

ID	Requirement Description	Priority
CLI-1	The `dgst` subcommand **must** be extended to support a new `--hmac` flag that enables HMAC mode.	Must
CLI-2	When `--hmac` is specified, the `--key KEY` argument **must** become mandatory. - The key **must** be provided as a hexadecimal string. - The key **must** be of arbitrary length (HMAC supports variable-length keys).	Must
CLI-3	The output format for HMAC **must** be: `HMAC_VALUE INPUT_FILE_PATH`	Must
CLI-4	The tool **should** support an optional `--verify FILE` flag that allows verification of an existing HMAC. - When `--verify` is used, the tool **must** compute the HMAC and compare it with the value in the specified file. - The tool **must** exit with code 0 for success and non-zero for failure, printing an appropriate message.	Should
CLI-5	**(Bonus)** The tool **may** support a `--cmac` flag to use AES-CMAC instead of HMAC.	Could

Example Invocations:

```
# Generate HMAC
$ cryptocore dgst --algorithm sha256 --hmac --key 00112233445566778899aabbccddeeff --input message.txt
> a1b2c3d4e5f60123456789012345678901234567890123456789012345678901234567890123  message.txt

# Verify HMAC (with --verify)
$ cryptocore dgst --algorithm sha256 --hmac --key 00112233445566778899aabbccddeeff --input message.txt --verify expected_hmac.txt
> [OK] HMAC verification successful

# Or verify by manual comparison
$ cryptocore dgst --algorithm sha256 --hmac --key 00112233445566778899aabbccddeeff --input message.txt > computed_hmac.txt
$ diff computed_hmac.txt expected_hmac.txt
```

3. HMAC Implementation

HMAC must be implemented from scratch using the student's SHA-256 implementation.

ID	Requirement Description	Priority
MAC-1	**HMAC must** be implemented from scratch by the student. - The implementation **must** follow RFC 2104 specification. - It **must** use the SHA-256 implementation from Sprint 4 as the underlying hash function.	Must
MAC-2	The HMAC implementation **must** correctly handle the key processing: - Keys longer than the block size (64 bytes for SHA-256) **must** be hashed first. - Keys shorter than the block size **must** be padded with zeros.	Must
MAC-3	The implementation **must** correctly compute: `HMAC(K, m) = H((K ⊕ opad) ‖ H((K ⊕ ipad) ‖ m))` where `H` is SHA-256, `opad` is 0x5c repeated, and `ipad` is 0x36 repeated.	Must
MAC-4	The implementation **must** support keys of arbitrary length.	Must
MAC-5	The implementation **must** process input files in chunks to maintain constant memory usage.	Must
MAC-6	**(Bonus) AES-CMAC may** be implemented from scratch following NIST SP 800-38B. - If implemented, it **must** use the AES primitive from earlier sprints. - It **must** correctly implement subkey generation and the CBC-MAC construction with final XOR.	Could

Expected HMAC Implementation Structure:

```
# Python: src/mac/hmac.py
class HMAC:
    def __init__(self, key, hash_function='sha256'):
        self.hash_function = get_hash_function(hash_function)  # Returns SHA256 instance
        self.block_size = 64  # bytes, for SHA-256
        self.key = self._process_key(key)

    def _process_key(self, key):
        # If key is longer than block size, hash it
        if len(key) > self.block_size:
            key = self.hash_function(key).digest()
        # If key is shorter, pad with zeros
        if len(key) < self.block_size:
            key = key + b'\x00' * (self.block_size - len(key))
        return key

    def _xor_bytes(self, a, b):
        return bytes(x ^ y for x, y in zip(a, b))

    def compute(self, message):
        # Create inner and outer pads
        ipad = self._xor_bytes(self.key, b'\x36' * self.block_size)
        opad = self._xor_bytes(self.key, b'\x5c' * self.block_size)

        # Inner hash: H((K ⊕ ipad) ‖ message)
        inner_hash = self.hash_function(ipad + message).digest()

        # Outer hash: H((K ⊕ opad) ‖ inner_hash)
        outer_hash = self.hash_function(opad + inner_hash).digest()

        return outer_hash.hex()
```

4. File I/O for MAC Operations

The MAC functionality must handle files efficiently and integrate with existing hash I/O.

ID	Requirement Description	Priority
IO-1	The tool **must** read the input file in binary mode, processing in chunks as with regular hashing.	Must
IO-2	When `--verify` is used, the tool **must** read the expected HMAC value from the specified file. - The expected HMAC file **should** be in the same format as generated by the tool (`HMAC_VALUE FILENAME`). - The tool **should** be flexible in parsing the expected HMAC file (ignore whitespace, filename).	Must
IO-3	If the `--output` flag (from Sprint 4) is used with HMAC, the tool **must** write the HMAC in the standard format.	Must

5. Testing & Verification

Comprehensive testing must ensure correctness and security properties of the MAC implementations.

ID	Requirement Description	Priority
TEST-1	**Known-Answer Tests:** The HMAC implementation **must** pass test vectors from RFC 4231. - Specifically, test cases 1-4 from Section 4.2 of RFC 4231.	Must
TEST-2	**Verification Test:** The tool **must** successfully verify HMACs that it generated. `$ cryptocore dgst --hmac --key <key> --input <file> --verify <hmac_file>` should succeed for valid pairs.	Must
TEST-3	**Tamper Detection - File:** The tool **must** detect when the input file has been modified. - Generate an HMAC for a file, then change one byte in the file. - Verification with the original key **must** fail.	Must
TEST-4	**Tamper Detection - Key:** The tool **must** detect when the wrong key is used. - Generate an HMAC with one key, then verify with a different key. - Verification **must** fail.	Must
TEST-5	**Key Size Tests:** The implementation **must** correctly handle various key sizes: - Keys shorter than block size (e.g., 16 bytes) - Keys equal to block size (64 bytes for SHA-256) - Keys longer than block size (e.g., 100 bytes)	Must
TEST-6	**Empty File Test:** The implementation **must** correctly compute HMAC for empty files.	Must
TEST-7	**Large File Test:** The implementation **must** correctly compute HMAC for files larger than available memory.	Must
TEST-8	**(Bonus)** If AES-CMAC is implemented, it **must** pass test vectors from NIST SP 800-38B.	Could

Example Test Commands:

```
# Test with RFC 4231 test vector 1
$ echo -n "Hi There" > test_file.txt
$ cryptocore dgst --algorithm sha256 --hmac --key 0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b --input test_file.txt
> Should produce: b0344c61d8db38535ca8afceaf0bf12b881dc200c9833da726e9376c2e32cff7
```

```
# Tamper detection test
$ echo "original content" > file.txt
$ cryptocore dgst --algorithm sha256 --hmac --key 00112233445566778899aabbccddeeff --input file.txt > original_hmac.txt

# Tamper with the file
$ echo "modified content" > file.txt
$ cryptocore dgst --algorithm sha256 --hmac --key 00112233445566778899aabbccddeeff --input file.txt --verify original_hmac.txt
> [ERROR] HMAC verification failed
> Exit code: 1

# Wrong key test
$ cryptocore dgst --algorithm sha256 --hmac --key 00112233445566778899aabbccddeeff --input file.txt --verify original_hmac.txt
> [ERROR] HMAC verification failed (different key)
```

Example Test Script for RFC 4231 Vectors:

```
# tests/test_hmac_vectors.py
from src.mac.hmac import HMAC

def test_rfc_4231():
    """Test HMAC with RFC 4231 test vectors"""
    test_cases = [
        {
            'key': '0b' * 20,  # 20 bytes of 0x0b
            'data': '4869205468657265',  # "Hi There"
            'expected': 'b0344c61d8db38535ca8afceaf0bf12b881dc200c9833da726e9376c2e32cff7'
        },
        {
            'key': '4a656665',  # "Jefe"
            'data': '7768617420646f2079612077616e7420666f72206e6f7468696e673f',  # "what do ya want for nothing?"
            'expected': '5bdcc146bf60754e6a042426089575c75a003f089d2739839dec58b964ec3843'
        }
    ]

    for i, test in enumerate(test_cases):
        key = bytes.fromhex(test['key'])
        data = bytes.fromhex(test['data'])

        hmac = HMAC(key, 'sha256')
        result = hmac.compute(data)

        assert result == test['expected'], f"RFC 4231 test case {i+1} failed: got {result}, expected {test['expected']}"
        print(f"RFC 4231 test case {i+1} passed")
```