Project: CryptoCore - Technical Requirements Document (Sprint 3)

Sprint Goal: Implement a secure source of randomness for keys and IVs.

1. Project Structure & Repository Hygiene

A new CSPRNG module must be created and integrated into the existing architecture.

ID	Requirement Description	Priority
STR-1	All requirements from previous Sprints (STR-1 to STR-4) **must** still be met.	Must
STR-2	A new, dedicated source file for the CSPRNG module **must** be created. - **Suggested Path:** `src/csprng.py` (Python) or `src/csprng.c` / `src/csprng.h` (C)	Must
STR-3	The `README.md` file **must** be updated to include: - Documentation for the new behavior when `--key` is omitted. - An example showing the tool generating and displaying a random key. - A note on the security properties of the CSPRNG used. - Instructions for running the NIST statistical test suite.	Must

2. Command-Line Interface (CLI) Parser

The CLI must be extended to support optional key generation.

ID	Requirement Description	Priority
CLI-1	The `--key` argument **must** become optional for encryption operations.	Must
CLI-2	If the `--key` argument is provided, the tool **must** use it, exactly as in previous sprints.	Must
CLI-3	If the `--key` argument is **not** provided during an **encryption** operation: - The tool **must** generate a secure random 16-byte (128-bit) key. - The generated key **must** be printed to the standard output (`stdout`) as a hexadecimal string, prefixed with a clear label. - The tool **must** then proceed with the encryption using this generated key.	Must
CLI-4	For **decryption** operations, the `--key` argument **must** remain mandatory. If it is not provided, the tool **must** print an error and exit.	Must
CLI-5	The tool **should** print a warning to `stderr` if a user-provided key is detected as weak (e.g., all zeros, sequential bytes). This is a "Should" to encourage best practices.	Should

Example Invocations:

```
# Encryption with automatic key generation
$ cryptocore --algorithm aes --mode ctr --encrypt --input plaintext.txt --output ciphertext.bin
> [INFO] Generated random key: 1a2b3c4d5e6f7890fedcba9876543210

# Decryption (key must always be provided)
$ cryptocore --algorithm aes --mode ctr --decrypt --key 1a2b3c4d5e6f7890fedcba9876543210 --input ciphertext.bin --output decrypted.txt
```

3. Cryptographically Secure Random Number Generation (CSPRNG)

The core of this sprint is the implementation of a secure, dedicated module for randomness.

ID	Requirement Description	Priority
RNG-1	A dedicated function (or set of functions) **must** be implemented in the new CSPRNG module.	Must
RNG-2	The primary function **must** have the signature `generate_random_bytes(num_bytes: int) -> bytes` (Python) or equivalent in C.	Must
RNG-3	The implementation **must** use a cryptographically secure source of randomness provided by the operating system or a vetted library: - **For Python:** The `os.urandom()` function **must** be used. `secrets` module is also acceptable but `os.urandom` is more direct. - **For C:** The `/dev/urandom` device **must** be used by opening the file and reading bytes, **or** the `RAND_bytes()` function from OpenSSL.	Must
RNG-4	The CSPRNG module **must not** use standard library random functions (e.g., Python's `random` module, C's `rand()`) as these are not cryptographically secure.	Must
RNG-5	The `generate_random_bytes` function **must** be integrated into the tool's core logic for: - Generating the encryption key when `--key` is not provided. - Generating all IVs for encryption operations (continuing from Sprint 2).	Must
RNG-6	The function **must** handle potential errors (e.g., cannot open `/dev/urandom`, `os.urandom` fails) by throwing a clear, actionable exception or error message.	Must

Example CSPRNG Code Snippets:

```
# Python: src/csprng.py
import os
```

```
def generate_random_bytes(num_bytes):
    """Generates a cryptographically secure random byte string."""
    return os.urandom(num_bytes)
```

```
// C: src/csprng.c (using OpenSSL)
#include <openssl/rand.h>

int generate_random_bytes(unsigned char *buffer, int num_bytes) {
    if (RAND_bytes(buffer, num_bytes) != 1) {
        // Error handling: RAND_bytes failed
        return -1;
    }
    return 0;
}
```

4. Key and IV Management

The management of cryptographic material must be updated to use the new CSPRNG.

ID	Requirement Description	Priority
KEY-1	During encryption, if a key is generated, it **must** be 16 bytes long for AES-128.	Must
KEY-2	The generated key **must** be printed to `stdout` in a clear, hexadecimal format exactly once, immediately after generation.	Must
KEY-3	The tool **must not** write the generated key to the output ciphertext file. The user is responsible for noting the key printed to the terminal.	Must
IV-1	The IV generation for all relevant modes (CBC, CFB, OFB, CTR) **must** use the new `generate_random_bytes` function.	Must
IV-2	The IV handling from Sprint 2 (prepending to ciphertext, reading from file/CLI for decryption) **must** remain unchanged.	Must

5. Testing & Verification

Testing must now include rigorous statistical analysis using the NIST test suite.

ID	Requirement Description	Priority
TEST-1	**Key Generation Test:** Running the tool for encryption without the `--key` option **must** result in a key being printed to the terminal and successful encryption. Subsequent decryption with the printed key **must** recover the original file.	Must
TEST-2	**Uniqueness Test:** A test script **must** be created that calls the `generate_random_bytes` function 1000 times to generate 1000 16-byte keys.	Must
	- The test **must** check that all 1000 keys are unique (no duplicates).	
	- The probability of a duplicate in 1000 samples is vanishingly small for a true CSPRNG, so any duplicate indicates a critical flaw.	
TEST-3	**NIST Statistical Test Suite:** The student **must** run the NIST Statistical Test Suite (STS) on the output of their CSPRNG.	Must
	- **Test Data:** Generate a sufficiently large binary file (recommended: 1-100 MB) filled with output from `generate_random_bytes`.	
	- **Tool:** Use the NIST STS tool (available from [NIST's website])	
	- **Procedure:** Follow the NIST STS documentation to run all 15 tests on the generated data.	
	- **Success Criteria:** The majority of tests should pass (p-value ≥ 0.01). A small number of failures is statistically expected, but widespread failures indicate a flawed RNG.	
	- **Documentation:** The test procedure and results **must** be documented in the `README.md` or a separate `TESTING.md` file.	
TEST-4	**Basic Distribution Test:** The test script **should** perform a basic entropy check, such as ensuring that the generated keys have a high Hamming weight on average (close to 50% bits set to '1').	Should
TEST-5	**Interoperability Test:** The tool must still pass all interoperability tests from Sprint 2. For these tests, the `--key` argument will be used, so the RNG is not involved.	Must

Example NIST Test Procedure:

```
# Example steps for NIST test suite (Python version)
# 1. Install NIST STS or download the C version and compile
# 2. Generate test data using the project's CSPRNG
$ python -c "from src.csprng import generate_random_bytes;
data = generate_random_bytes(1000000);
open('random_test_data.bin', 'wb').write(data)"

# 3. Run NIST STS on the generated file
$ ./assess 1000000
# (Follow NIST STS interactive prompts to specify the test file)
# 4. Analyze results in the generated reports
```

Example Test Script for Uniqueness and Basic Statistics (Python):

```
# tests/test_csprng.py
from src.csprng import generate_random_bytes

def test_key_uniqueness():
    key_set = set()
    num_keys = 1000
    for _ in range(num_keys):
        key = generate_random_bytes(16)
```

```python
        key_hex = key.hex()
        # Check for uniqueness
        assert key_hex not in key_set, f"Duplicate key found: {key_hex}"
        key_set.add(key_hex)
    print(f"Successfully generated {len(key_set)} unique keys.")

def test_nist_preparation():
    """Generate a large random file for NIST testing"""
    total_size = 10_000_000  # 10 MB
    with open('nist_test_data.bin', 'wb') as f:
        bytes_written = 0
        while bytes_written < total_size:
            chunk_size = min(4096, total_size - bytes_written)
            random_chunk = generate_random_bytes(chunk_size)
            f.write(random_chunk)
            bytes_written += chunk_size
    print(f"Generated {bytes_written} bytes for NIST testing in 'nist_test_data.bin'")
```