Test Strategy for Cryptocurrency Demo Project

**Overview**

This cryptocurrency project include functions for hashing and conversion between hexadecimal and binary representations. A robust testing strategy is essential to ensure the security and accuracy of these cryptographic operations.

**Objectives**

1. **Functional Correctness:** Verify that the cryptographic functions produce accurate and consistent results.
2. **Security:** Ensure that the cryptographic functions meet security standards and resist attacks.
3. **Performance:** Evaluate the performance of cryptographic operations, especially with large data sets.
4. **Compatibility:** Confirm that the cryptographic functions work across different environments and dependencies.

**Test Levels**

1. **Unit Testing:**
   * **Components:** Individual cryptographic functions (**cryptographic\_hash**, **hex\_to\_binary**).
   * **Objectives:**
     + Validate the correctness of hashing and conversion functions.
     + Ensure functions handle various input scenarios, including edge cases.
2. **Integration Testing:**
   * **Components:** Interaction between different parts of the cryptographic scripts.
   * **Objectives:**
     + Validate the integration of hashing and conversion functions in the overall code.
     + Confirm proper data flow between components.
3. **System Testing:**
   * **Components:** Cryptographic scripts as a whole.
   * **Objectives:**
     + Verify the overall functionality of the cryptographic scripts.
     + Evaluate the system's behaviour in different scenarios, including concurrent usage.

**Test Types**

**In scope**

1. **Functional Testing:**
   * **Scenarios:**
     + Hash consistency for the same input.
     + Order independence in hashing.
     + Accuracy in hexadecimal to binary conversion.
   * **Test Methods:**
     + Test cases for known inputs with expected outputs.
2. **Security Testing:**
   * **Scenarios:**
     + Input validation for malicious data.
     + Resistance against common attacks (e.g., collision attacks).
   * **Test Methods:**
     + Fuzz testing with random inputs.
     + Security analysis of the code.
3. **Performance Testing:**
   * **Scenarios:**
     + Large data sets for hashing and conversion.
     + Time taken for different input sizes.
   * **Test Methods:**
     + Measure execution time for various input sizes.
     + Evaluate system resource consumption.
4. **Compatibility Testing:**
   * **Scenarios:**
     + Run tests on different operating systems.
     + Verify compatibility with different Python versions.
   * **Test Methods:**
     + Run tests on diverse environments.
     + Check dependencies for version compatibility.

**Automation**

1. **Unit Testing Automation:**
   * Use a testing framework (e.g., pytest) for automated unit tests.
   * Implement test cases for each function.
2. **Integration Testing Automation:**
   * Develop automated tests that cover the interaction between cryptographic functions.
3. **System Testing Automation:**
   * Automate tests for overall system functionality.
   * Include both positive and negative test scenarios.

**Test Data**

1. **Functional Testing Data:**
   * Generate test data with known inputs and expected outputs.
   * Include edge cases and scenarios with special characters.
2. **Security Testing Data:**
   * Use random and malicious inputs to test resistance against attacks.
3. **Performance Testing Data:**
   * Generate large datasets for performance testing.
   * Vary input sizes for hashing and conversion.

**Test Environment**

1. **Programming Language:**
   * Python (compatible versions).
2. **Dependencies:**
   * Ensure compatibility with specified dependency versions.
3. **Operating Systems:**
   * Test on different operating systems, including Windows, Linux, and macOS.

**Exit Criteria**

1. **Functional Correctness:**
   * All unit tests pass without failures.
   * Integration tests validate the correct interaction between components.
2. **Security:**
   * No security vulnerabilities are identified.
   * Malicious input does not compromise the system.
3. **Performance:**
   * Cryptographic functions perform efficiently for various input sizes.
4. **Compatibility:**
   * Code runs without errors on all specified platforms.
   * Dependencies are compatible with the specified versions.

**Risks and Mitigations**

1. **Security Risks:**
   * Regular security audits and code reviews to identify vulnerabilities.
2. **Performance Risks:**
   * Optimize code for performance bottlenecks identified during testing.
3. **Compatibility Risks:**
   * Ensure continuous integration tests on different environments.

**Review and Approval**

The test strategy is subject to review and approval by the development and testing teams. Adjustments may be made based on feedback and changes in project requirements. Once approved, the testing activities will be carried out in alignment with this strategy.