**PROJECT REPORT**

**ON**

**TESTING AND VALIDATION ELECTRIC VEHICLE COMMUNICATION SYSTEM**

**Table of Content**

1. Conduct unit testing and integration testing.

2. Validate system performance and reliability.

3. Perform fault injection testing for error handling.

4. Conduct security testing and vulnerability assessment.

**HOW TO CONDUCT UNIT TESTING AND INTEGRATION TESTING IN EV VEHICLES**

#include <CUnit/Basic.h>

/\* Function to be tested \*/

int add(int a, int b) {

return a + b;

}

/\* Test case \*/

void test\_add(void) {

CU\_ASSERT(add(2, 3) == 5);

CU\_ASSERT(add(-1, 1) == 0);

CU\_ASSERT(add(-1, -1) == -2);

}

int main() {

CU\_initialize\_registry();

CU\_pSuite suite = CU\_add\_suite("Suite\_1", 0, 0);

CU\_add\_test(suite, "test\_add", test\_add);

CU\_basic\_set\_mode(CU\_BRM\_VERBOSE);

CU\_basic\_run\_tests();

CU\_cleanup\_registry();

return 0;

}

 #include <CUnit/Basic.h>: This includes the CUnit testing framework.

 **Function** add(int a, int b): A simple function that returns the sum of two integers.

 **Test Case** test\_add(void): A function to test the add function.

* CU\_ASSERT(add(2, 3) == 5): Asserts that adding 2 and 3 equals 5.
* CU\_ASSERT(add(-1, 1) == 0): Asserts that adding -1 and 1 equals 0.
* CU\_ASSERT(add(-1, -1) == -2): Asserts that adding -1 and -1 equals -2.

 main() **function**:

* CU\_initialize\_registry(): Initializes the CUnit test registry.
* CU\_pSuite suite = CU\_add\_suite("Suite\_1", 0, 0): Adds a test suite named "Suite\_1".
* CU\_add\_test(suite, "test\_add", test\_add): Adds the test\_add test to the suite.
* CU\_basic\_set\_mode(CU\_BRM\_VERBOSE): Sets the test mode to verbose to provide detailed output.
* CU\_basic\_run\_tests(): Runs the tests.
* CU\_cleanup\_registry(): Cleans up the test registry.

Integration Testing:

#include <CUnit/Basic.h>

/\* Functions to be tested \*/

int multiply(int a, int b) {

return a \* b;

}

int square\_sum(int a, int b) {

return multiply(a, a) + multiply(b, b);

}

/\* Integration test \*/

void test\_square\_sum(void) {

CU\_ASSERT(square\_sum(2, 3) == 13);

CU\_ASSERT(square\_sum(-1, 1) == 2);

CU\_ASSERT(square\_sum(-2, -2) == 8);

}

int main() {

CU\_initialize\_registry();

CU\_pSuite suite = CU\_add\_suite("Suite\_1", 0, 0);

CU\_add\_test(suite, "test\_square\_sum", test\_square\_sum);

CU\_basic\_set\_mode(CU\_BRM\_VERBOSE);

CU\_basic\_run\_tests();

CU\_cleanup\_registry();

return 0;

}

 #include <CUnit/Basic.h>: This includes the CUnit testing framework.

 **Function** multiply(int a, int b): Multiplies two integers and returns the result.

 **Function** square\_sum(int a, int b): Computes the sum of the squares of two integers.

* return multiply(a, a) + multiply(b, b): Uses the multiply function to square a and b, then adds the results.

 **Integration Test** test\_square\_sum(void): A function to test the square\_sum function.

* CU\_ASSERT(square\_sum(2, 3) == 13): Asserts that the square sum of 2 and 3 equals 13.
* CU\_ASSERT(square\_sum(-1, 1) == 2): Asserts that the square sum of -1 and 1 equals 2.
* CU\_ASSERT(square\_sum(-2, -2) == 8): Asserts that the square sum of -2 and -2 equals 8.

 main() **function**:

* CU\_initialize\_registry(): Initializes the CUnit test registry.
* CU\_pSuite suite = CU\_add\_suite("Suite\_1", 0, 0): Adds a test suite named "Suite\_1".
* CU\_add\_test(suite, "test\_square\_sum", test\_square\_sum): Adds the test\_square\_sum test to the suite.
* CU\_basic\_set\_mode(CU\_BRM\_VERBOSE): Sets the test mode to verbose to provide detailed output.
* CU\_basic\_run\_tests(): Runs the tests.
* CU\_cleanup\_registry(): Cleans up the test.

**2. Validate System Performance and Reliability**

* **Performance Testing:**
* Performance testing in C can be done by measuring the execution time:

#include <stdio.h>

#include <time.h>

void performance\_test() {

clock\_t start\_time = clock();

// Code to be tested

for (int i = 0; i < 1000000; i++) {

// Simulating a task

}

clock\_t end\_time = clock();

double time\_spent = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

printf("Execution time: %f seconds\n", time\_spent);

}

int main() {

performance\_test();

return 0;

}

 **Include Header Files:**

c

#include <stdio.h>

#include <time.h>

* #include <stdio.h>: This includes the Standard Input Output library, which is necessary for the printf function.
* #include <time.h>: This includes the time library, which provides functions to measure time.

 **Function** performance\_test**:**

c

void performance\_test() {

clock\_t start\_time = clock();

* clock\_t start\_time = clock();: This captures the current processor time at the start of the function using the clock() function. clock\_t is a data type used to store time values.

 **Simulated Task:**

// Code to be tested

for (int i = 0; i < 1000000; i++) {

// Simulating a task

}

* for (int i = 0; i < 1000000; i++): This is a loop that runs 1,000,000 times to simulate a task. The content of the loop could be replaced with the actual code you want to test.

 **End Time Measurement:**

clock\_t end\_time = clock();

* clock\_t end\_time = clock();: This captures the processor time at the end of the function.

 **Calculate and Print Execution Time:**

double time\_spent = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

printf("Execution time: %f seconds\n", time\_spent);

* double time\_spent = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;: This calculates the elapsed time in seconds. CLOCKS\_PER\_SEC is a constant that represents the number of clock ticks per second.
* printf("Execution time: %f seconds\n", time\_spent);: This prints the execution time to the console.

 **Main Function:**

int main() {

performance\_test();

return 0;

}

* int main(): The main function where the performance\_test function is called.
* performance\_test();: This calls the performance\_test function to measure and print the execution time.
* return 0;: This indicates that the program executed Sucessfully.

**Reliability Testing:**

* Reliability testing involves running the code multiple times:

#include <assert.h>

#include <stdlib.h>

int add(int a, int b) {

return a + b;

}

void reliability\_test() {

for (int i = 0; i < 1000; i++) {

int a = rand() % 100;

int b = rand() % 100;

int result = add(a, b);

assert(result == a + b);

}

}

int main() {

reliability\_test();

return 0;

}

c

#include <assert.h>

#include <stdlib.h>

int add(int a, int b) {

return a + b;

}

void reliability\_test() {

for (int i = 0; i < 1000; i++) {

int a = rand() % 100;

int b = rand() % 100;

int result = add(a, b);

assert(result == a + b);

}

}

int main() {

reliability\_test();

return 0;

}

**Detailed Explanation:**

1. **Include Header Files:**

#include <assert.h>

#include <stdlib.h>

* + #include <assert.h>: This includes the assert library, which provides the assert function used for debugging.
  + #include <stdlib.h>: This includes the standard library, which provides functions such as rand for generating random numbers.

1. **Function** add**:**

int add(int a, int b) {

return a + b;

}

* + This function takes two integers as input and returns their sum.

1. **Function** reliability\_test**:**

void reliability\_test() {

for (int i = 0; i < 1000; i++) {

int a = rand() % 100;

int b = rand() % 100;

int result = add(a, b);

assert(result == a + b);

}

}

* + for (int i = 0; i < 1000; i++): This loop runs 1,000 times to simulate multiple test cases.
  + int a = rand() % 100: Generates a random integer a between 0 and 99.
  + int b = rand() % 100: Generates a random integer b between 0 and 99.
  + int result = add(a, b): Calls the add function with the generated integers a and b and stores the result.
  + assert(result == a + b): Asserts that the result of the add function is equal to the sum of a and b. If this condition is false, the program will terminate with an assertion failure. This ensures that the add function works correctly across a variety of input values.

1. **Main Function:**

int main() {

reliability\_test();

return 0;

}

* + int main(): The main function where the reliability\_test function is called.
  + reliability\_test();: This calls the reliability\_test function to run the reliability tests.
  + return 0;: This indicates that the program executed successfully.

### 3. Perform Fault Injection Testing for Error Handling

* **Fault Injection Testing:**
* Fault injection in C can be used to test error handling:

#include <stdio.h>

#include <stdlib.h>

int divide(int a, int b) {

if (b == 0) {

fprintf(stderr, "Division by zero error!\n");

exit(1);

}

return a / b;

}

void fault\_injection\_test() {

if (divide(10, 0)) {

printf("Error handling test passed\n");

}

}

int main() {

fault\_injection\_test();

return 0;

}

 **Include Header Files:**

c

#include <stdio.h>

#include <stdlib.h>

* #include <stdio.h>: This includes the Standard Input Output library, which is necessary for functions like printf and fprintf.
* #include <stdlib.h>: This includes the Standard Library, which is necessary for functions like exit.

 **Function** divide**:**

int divide(int a, int b) {

if (b == 0) {

fprintf(stderr, "Division by zero error!\n");

exit(1);

}

return a / b;

}

* int divide(int a, int b): This function takes two integers, a and b.
* if (b == 0): Checks if b is zero.
* fprintf(stderr, "Division by zero error!\n"): Prints an error message to the standard error stream if b is zero.
* exit(1): Exits the program with a status of 1 if b is zero, indicating an error.
* return a / b: Returns the result of dividing a by b if b is not zero.

 **Function** fault\_injection\_test**:**

void fault\_injection\_test() {

if (divide(10, 0)) {

printf("Error handling test passed\n");

}

}

* void fault\_injection\_test(): This function is designed to test the error handling of the divide function.
* if (divide(10, 0)): Calls the divide function with 10 and 0 as arguments. Since b is zero, the divide function will print an error message and exit the program. Therefore, the code inside the if statement will never execute.
* This is a deliberate fault injection to ensure that the divide function correctly handles a division by zero.

 **Main Function:**

int main() {

fault\_injection\_test();

return 0;

}

* int main(): The main function where the fault\_injection\_test function is called.
* fault\_injection\_test();: Calls the fault\_injection\_test function to perform the fault injection test.
* return 0;: This indicates that the program executed successfully (although it will exit with status 1 if a division by zero is attempted).

**4. Conduct Security Testing and Vulnerability Assessment**

#include <stdio.h>

#include <string.h>

int is\_vulnerable(const char\* query) {

const char\* dangerous\_input = "' OR '1'='1";

if (strstr(query, dangerous\_input)) {

return 1;

}

return 0;

}

void security\_test()

{ const char\* test\_query = "SELECT \* FROM users WHERE username = 'admin' AND password = '' OR '1'='1'";

if (is\_vulnerable(test\_query)) {

printf("Potential SQL Injection Vulnerability detected!\n");

}

}

int main()

{

security\_test();

return 0;

}

 **Include Header Files:**

#include <stdio.h>

#include <string.h>

* #include <stdio.h>: This includes the Standard Input Output library, which is necessary for functions like printf.
* #include <string.h>: This includes the string library, which provides functions like strstr for string manipulation.

 **Function** is\_vulnerable**:**

int is\_vulnerable(const char\* query) {

const char\* dangerous\_input = "' OR '1'='1";

if (strstr(query, dangerous\_input)) {

return 1;

}

return 0;

}

* int is\_vulnerable(const char\* query): This function takes a SQL query as input and checks if it contains a known dangerous pattern.
* const char\* dangerous\_input = "' OR '1'='1";: This defines a dangerous SQL injection input pattern.
* if (strstr(query, dangerous\_input)): Checks if the dangerous input pattern exists within the query. strstr is a function that searches for a substring within a string.
* return 1;: If the pattern is found, the function returns 1, indicating a potential vulnerability.
* return 0;: If the pattern is not found, the function returns 0, indicating no vulnerability detected.

 **Function** security\_test**:**

void security\_test() {

const char\* test\_query = "SELECT \* FROM users WHERE username = 'admin' AND password = '' OR '1'='1'";

if (is\_vulnerable(test\_query)) {

printf("Potential SQL Injection Vulnerability detected!\n");

}

}

* void security\_test(): This function is designed to test for SQL injection vulnerabilities.
* const char\* test\_query = "SELECT \* FROM users WHERE username = 'admin' AND password = '' OR '1'='1'";: Defines a test SQL query that contains a known SQL injection pattern.
* if (is\_vulnerable(test\_query)): Calls the is\_vulnerable function with the test query. If the function returns 1, it means the query contains the dangerous pattern.
* printf("Potential SQL Injection Vulnerability detected!\n"): Prints a message indicating that a potential SQL injection vulnerability has been detected.

 **Main Function:**

int main() {

security\_test();

return 0;

}

* int main(): The main function where the security\_test function is called.
* security\_test();: Calls the security\_test function to perform the security test.
* return 0;: Indicates that the program executed successfully.

**Conclusion of this entire report**

**1. Unit Testing and Integration Testing**

* **Unit Testing**: By testing individual modules like the Battery Management System (BMS), Vehicle Control Unit (VCU), and communication protocols in isolation, the foundational functionality of the system is validated. The use of automated testing frameworks ensures repeatability, consistency, and early detection of errors.
* **Integration Testing**: The interaction between modules and systems (e.g., VCU to Motor Controller, EV to Charger Communication) is rigorously tested, ensuring that the data flow, synchronization, and combined functionalities work as intended. Hardware-in-the-Loop (HIL) and software simulators enhance the testing accuracy for real-world scenarios.

**2. System Performance and Reliability Validation**

* Performance testing evaluates the EV system under various workloads to identify and eliminate bottlenecks. Metrics like latency, throughput, and response times are validated to ensure the system operates optimally under normal and peak conditions.
* Reliability testing demonstrates the system’s ability to function consistently over time, even under stress, by simulating extended operational durations or fluctuating environmental conditions.

**3. Fault Injection Testing**

* Fault injection testing validates the system's error-handling capabilities by simulating failures such as communication breakdowns, hardware malfunctions, or invalid data. The testing process ensures the system can detect, recover from, and mitigate such errors without catastrophic failure, thereby improving its robustness and fault tolerance.

**4. Security Testing and Vulnerability Assessment**

* Security testing identifies vulnerabilities and ensures data protection, integrity, and confidentiality across the EV communication system. By addressing threats like unauthorized access or data breaches, the system is safeguarded against external and internal attacks.
* Vulnerability assessments highlight potential weak points, enabling preemptive action to strengthen the overall security architecture.