# Software Testing Assignment # 1

## Faulty Programs Analysis

Ping-Chen Chung

October 2, 2025

**GitHub Repository:** `https://github.com/Quisette/NYCU-Software-Testing-HW1`

## Contents

# 1 Part 1: JavaScript Programs - Fault Analysis

This section analyzes four faulty JavaScript programs, identifying faults, proposing fixes, and exploring test cases that execute faults, reach error states, and produce failures.

## 1.1 Program 1: `findLast(x, y)`

### Problem Description

The function should find the last index of element `y` in array `x`.

#### (a) Fault Explanation and Fix

The fault is in the `for` loop's continuation condition. The condition `i > 0` causes the loop to stop when `i` is 0, so the element at index 0 of the array is never checked. The function will incorrectly return -1 if the element being searched for is only present at index 0.

**Modification:**

```
// Original (FAULTY):
for (let i = x.length - 1; i > 0; i--)

// Corrected:
for (let i = x.length - 1; i >= 0; i--)
```

#### (b) Test Case That Does Not Execute the Fault

This is not strictly possible, as any call with a non-empty array will execute the line of code containing the fault (the `for` loop statement). However, it is possible to provide a test case where the fault is executed but does not cause the program to fail.

**Example:** `x = [1, 2, 3]`, `y = 2`. The program correctly returns `1`.

#### (c) Test Case That Executes Fault But No Error State

- **Test Case:** `x = [10, 20, 30]`, `y = 20`
- **Explanation:** The faulty code in the `for` loop is executed. However, the value `20` is found at index `1`. The function correctly returns `1` and terminates before the loop counter `i` reaches `0`. Because the search is successful before the boundary condition is tested, the fault does not lead to an incorrect internal state.
- **Expected Output:** `1`
- **Actual Output:** `1`

#### (d) Test Case With Error State But No Failure

### Not Possible.

For this function, an error state occurs when the loop completes without finding `y` because `y` is at index 0. At this moment, the program's internal state is incorrect because it has failed to find the element. This error state immediately leads to the function returning `-1`. Since the incorrect return value *is* the external incorrect behavior, the error state directly causes a failure. There are no subsequent operations that could mask or correct this error.

#### (e) First Error State

Not applicable, as a test case for (d) is not possible.

## 1.2 Program 2: `lastZero(x)`

### Problem Description

The function should find the last index of zero in array `x`.

### (a) Fault Explanation and Fix

The function is named `lastZero`, but its implementation finds the *first* index of zero. The fault is that the `for` loop iterates forward from the beginning of the array (`i = 0; i < x.length; i++`) and returns immediately upon finding the first match.

**Modification:**

```
// Original (FAULTY):
for (let i = 0; i < x.length; i++)

// Corrected:
for (let i = x.length - 1; i >= 0; i--)
```

### (b) Test Case That Does Not Execute the Fault

This is not strictly possible, as any call with a non-empty array will execute the faulty `for` loop. However, we can provide a case where the fault does not lead to a failure, such as when the first zero is also the last zero: `x = [5, 1, 0, 8]`.

### (c) Test Case That Executes Fault But No Error State

- **Test Case:** `x = [1, 5, 0, 8]`
- **Explanation:** The faulty forward-iterating loop is executed. However, since there is only one `0` in the array, the index of the first `0` is the same as the index of the last `0`. The program correctly returns `2`. The internal state is never incorrect, so no error state is reached.
- **Expected Output:** `2`
- **Actual Output:** `2`

### (d) Test Case With Error State But No Failure

**Not Possible.**

An error state occurs when the function finds a `0` at an early index `i` and is about to return `i`, while the true last `0` exists at a later index `j`. This incorrect internal decision (to return `i`) is inseparable from the external failure (returning the wrong index). The error cannot be corrected before the function terminates.

### (e) First Error State

Not applicable, as a test case for (d) is not possible.

## 1.3 Program 3: `countPositive(x)`

### Problem Description

The function should count strictly positive (greater than 0) elements in array `x`.

#### (a) Fault Explanation and Fix

The function is intended to count "positive" elements, which are strictly greater than 0. The fault is that the condition `x[i] >= 0` incorrectly includes `0` in the count.

**Modification:**

```
// Original (FAULTY):
if (x[i] >= 0)

// Corrected:
if (x[i] > 0)
```

#### (b) Test Case That Does Not Execute the Fault

This is not strictly possible, as the faulty `if` condition is executed for each array element. However, we can provide a case that does not expose the fault by using an array that does not contain `0`: `x = [-2, 5, 10]`.

#### (c) Test Case That Executes Fault But No Error State

- **Test Case:** `x = [-5, 1, 9]`
- **Explanation:** The faulty condition `x[i] >= 0` is executed for each element. Because the input array does not contain `0`, the behavior of `>= 0` is identical to `> 0` for every element. The internal state variable `count` is always correct throughout the execution, so no error state is entered.
- **Expected Output:** `2`
- **Actual Output:** `2`

#### (d) Test Case With Error State But No Failure

**Not Possible.**

An error state occurs when the program processes a `0` and incorrectly increments `count`. At this point, the `count` variable is one greater than its correct value. Since the `count` variable is only ever incremented, this error cannot be corrected. The incorrect value will persist and be returned at the end, directly causing a failure.

#### (e) First Error State

Not applicable, as a test case for (d) is not possible.

### 1.4 Program 4: `oddOrPos(x)`

#### Problem Description

The function should count elements that are either odd or positive.

#### (a) Fault Explanation and Fix

The fault is in the logic used to identify odd numbers. The condition `x[i] % 2 === 1` fails for negative odd numbers. In JavaScript, the result of the modulo operator (`%`) on a negative number is negative (e.g., `-3 % 2` is `-1`), so `-1 === 1` evaluates to false.

**Modification:**

```
// Original (FAULTY):
if (x[i] % 2 === 1 || x[i] > 0)

// Corrected:
if (x[i] % 2 !== 0 || x[i] > 0)
```

### (b) Test Case That Does Not Execute the Fault

This is not strictly possible, as the faulty `if` condition is always executed. However, we can provide a test case that does not expose the fault by avoiding negative odd numbers: `x = [-2, 1, 4]`.

### (c) Test Case That Executes Fault But No Error State

- **Test Case:** `x = [-2, 3, 4]`
- **Explanation:** The faulty condition `x[i] % 2 === 1` is executed. Since the input contains no negative odd numbers, the faulty logic behaves correctly for all elements. `3 % 2` is `1`, so it is counted. `-2` is not counted. `4` is counted because it's positive. The `count` variable remains correct throughout the execution, so no error state is reached.
- **Expected Output:** `2`
- **Actual Output:** `2`

### (d) Test Case With Error State But No Failure

**Not Possible.**

An error state occurs when the program processes a negative odd number (like `-3`) and fails to increment `count`. At this point, the `count` variable is less than its correct value. As `count` is never decremented, it is impossible for it to "catch up" or be corrected later in the execution. This persistent error in the internal state will inevitably lead to an incorrect final return value, which is a failure.

### (e) First Error State

Not applicable, as a test case for (d) is not possible.

# 2 Part 2: Multi-Language Programs - Verification Framework

This section documents the comprehensive analysis and automated verification framework built for five faulty programs demonstrating real-world software defects: deadlock, memory leaks, race conditions, buffer overflows, and const violations.

## 2.1 Overview

This document details all issues discovered in the software testing homework assignment, including incomplete source files, verifier script problems, and the solutions implemented to create a comprehensive automated testing framework.

## 2.2 Issues Found in Verifier Script (`verifier.cpp`)

### Issue 1: Incorrect Compiler Flag

**Location:** Line 17

**Problem:** Used `-Wno-fpermissive` which disables permissive mode warnings, causing compilation errors for intentionally faulty code.

```
// BEFORE (WRONG):
std::string command = "g++-15 -g -pthread " + source + " -o " + output
    + " -isysroot $(xcrun --show-sdk-path) -Wno-fpermissive ";

// AFTER (FIXED):
std::string command = "g++-15 -std=c++11 -g -pthread " + source + " -o " + output
    + " -isysroot $(xcrun --show-sdk-path) -fpermissive ";
```

**Impact:** ProfileUpdater.cpp and MatrixProcessor.cpp failed to compile.
**Solution:** Changed to `-fpermissive` to allow intentionally faulty conversions to compile.

### Issue 2: Memory Leak Detection - Wrong Output Redirection

**Location:** Line 59

**Problem:** Redirected only stderr (2>) but `leaks` tool outputs to stdout.

```
// BEFORE (WRONG):
std::system("leaks -atExit -- ./data_processer 2> leaks_output.txt");

// AFTER (FIXED):
std::system("leaks -atExit -- ./data_processer > evidence/leaks_output.txt 2>&1");
```

**Impact:** Memory leak output was not captured, test always failed.
**Solution:** Redirect both stdout and stderr to capture full `leaks` output.

### Issue 3: Memory Leak Detection Pattern Too Strict

**Location:** Lines 63-64

**Problem:** Only looked for “leaked bytes” pattern, missing “leaks for” pattern.

```
// BEFORE (WRONG):
if (output.find("leaked bytes") != std::string::npos &&
    output.find("0 leaks for 0 bytes") == std::string::npos)

// AFTER (FIXED):
if ((output.find("leaked bytes") != std::string::npos ||
     output.find("leaks for") != std::string::npos) &&
    output.find("0 leaks for 0") == std::string::npos)
```

**Impact:** Failed to detect leaks with format "20000 leaks for 480000 total leaked bytes."
**Solution:** Accept either pattern and generalize the "no leak" check.

**Issue 4: No Evidence File Organization**

**Problem:** Test output files scattered in root directory, no preservation after cleanup.
**Solution Implemented:**

1. Created `evidence/` directory to store all test outputs
2. Updated all test functions to write to `evidence/[test_name]_output.txt`
3. Modified cleanup to preserve evidence files
4. Added evidence file notification in PASS messages

Evidence files created:

- `evidence/deadlock_output.txt`
- `evidence/leaks_output.txt`
- `evidence/race_output.txt`
- `evidence/overflow_output.txt`
- `evidence/const_violation_output.txt`

**Issue 5: Testing Incomplete Source Files**

**Problem:** Verifier attempted to compile and test incomplete source files (missing main functions, syntax errors).
**Solution:** Created separate test implementation files in `tests/` directory:

- `tests/test_deadlock.py` - Complete deadlock test with main function
- `tests/test_buffer_overflow.cpp` - Complete buffer overflow test
- `tests/test_const_violation.cpp` - Complete const violation test with global const data

This approach:

- Keeps original buggy source files unchanged
- Provides complete test implementations
- Safely handles crashes without affecting verifier
- Makes tests reproducible

## 2.3 Issues Found in Source Files

**Program 1: ResourceScheduler.py - Deadlock**

**Issues Found:**

1. **Line 19:** Syntax error - incomplete assignment

```
task_queue = # SYNTAX ERROR
```

2. **Lines 79-81:** Empty main block

```
if __name__ == "__main__":
    # Empty - no code to start threads
```

**Solution:** Created `tests/test_deadlock.py` with complete implementation including:

- Fixed `task_queue = []`
- Complete main block that starts both threads
- Timeout mechanism to detect deadlock

**Verification Evidence:**

```
# From evidence/deadlock_output.txt
2025-10-02 01:16:23,456 | Thread-A | Thread-A is starting.
2025-10-02 01:16:23,456 | Thread-A | Thread-A attempting to acquire lock on Resource A...
2025-10-02 01:16:23,456 | Thread-A | Thread-A acquired lock on Resource A.
                                       Waiting for Resource B...
2025-10-02 01:16:23,457 | Thread-B | Thread-B is starting.
2025-10-02 01:16:23,457 | Thread-B | Thread-B attempting to acquire lock on Resource B...
2025-10-02 01:16:23,457 | Thread-B | Thread-B acquired lock on Resource B.
                                       Waiting for Resource A...
# Program hangs here - DEADLOCK detected by timeout
```

### Program 2: ProfileUpdater.cpp - Buffer Overflow

**Issues Found:**

1. **Line 11:** Misleading comment

```cpp
char username[1]; // Buffer of 20 characters <- WRONG! Only 1 byte
```

2. **Line 20:** Wrong memset usage

```cpp
memset(profile_status, 0, sizeof(profile_status)); // Invalid conversion char to
    void*
```

   Should be: `memset(&profile_status, 0, sizeof(profile_status));` or just `profile_status = '\0';`

3. **Lines 48-50:** Empty main function

```cpp
int main() {
    // trigger the fault
    return 0;
}
```

**Solution:** Created `tests/test_buffer_overflow.cpp` with:

- Corrected memset call
- Complete test that creates profile, sets fields, triggers overflow, shows corruption
- Long string that overflows 1-byte buffer

**Verification Evidence:**

```
# From evidence/overflow_output.txt
--- Profile Before Update ---
Username:
User ID: 12345
Status: A
Is Active: Yes
Last Login: 2024
----------------------------

Attempting to update with a malicious, oversized username...

--- Profile After Malicious Update ---
Username: ThisIsAVeryLongUsernameThatWillCauseABufferOverflow
User ID: 1936269938  # CORRUPTED! Was 12345
Status: <garbage>     # CORRUPTED! Was 'A'
Is Active: Yes
Last Login: 1702129518  # CORRUPTED! Was 2024
```

**Result:** Memory corruption confirmed - adjacent struct members overwritten.

### Program 3: MatrixProcessor.cpp - Const Violation

**Issues Found:**

1. **Line 27:** Invalid syntax

```cpp
non_const_matrix = 999; // Invalid conversion from 'int' to 'int**'
```

   Should be: `non_const_matrix[0][0] = 999;`

2. **Lines 30-34:** Empty main function

```cpp
int main() {
    // trigger the fault
    return 0;
}
```

**Solution:** Created `tests/test_const_violation.cpp` with:

- **Critical fix:** Used global `static const` arrays instead of local const arrays
- Local const arrays are on the stack (writable)
- Global const arrays are in read-only memory segment
- Attempting to modify triggers segmentation fault

```cpp
// Global const data - placed in read-only memory
static const int row0[] = {10, 20, 30};
static const int row1[] = {40, 50, 60};

int main() {
    const int* const matrix[] = {row0, row1};
    process_matrix(matrix, 2, 3); // CRASH!
}
```

**Verification Evidence:**

```
# From evidence/const_violation_output.txt
Value before calling function: matrix[0][0] = 10

Calling function that will attempt to modify const data...
Inside process_matrix function.
Processing a 2x3 matrix.
Attempting to modify const data (undefined behavior)...
# SEGMENTATION FAULT - Program crashed (exit code 138)
```

**Result:** Program crashed as expected when attempting to write to read-only memory.

### Program 4: LoggingSystem.cpp - Race Condition

**Status:** ✓Complete - No issues found

This file was complete and functional. The race condition is inherent in the design:

```cpp
void processLogs(int thread_id, int num_logs_to_process) {
    for (int i = 0; i < num_logs_to_process; ++i) {
        simulateLogProcessing();
        total_logs_processed++; // NO MUTEX - Race condition here
    }
}
```

**Verification Evidence:**

```
# From evidence/race_output.txt
Starting logging system test...
Processing logs across 10 threads...
All threads completed.

Expected total logs: 100000
Final count of logs processed: 98320  # Lost 1680 updates due to race!
```

**Result:** Race condition confirmed - final count consistently less than expected.

### Program 5: DataProcessor.cpp - Memory Leak

**Issues Found:** **Line 39:** Wrong array allocation

```cpp
newRecord->data_buffer = new char; // Only allocates 1 byte!
```

Should be: `newRecord->data_buffer = new char[bufferSize];`

However, this doesn't affect leak detection since the memory is still leaked either way.

**Status:** ✓Functionally Complete

The main function properly triggers the leak by calling `startDataIngestion()` which allocates 10,000 records without cleanup.

**Verification Evidence:**

```
# From evidence/leaks_output.txt
Process 65191: 20190 nodes malloced for 580 KB
Process 65191: 20000 leaks for 480000 total leaked bytes.

STACK OF 10000 INSTANCES OF 'ROOT LEAK: <malloc in processLargeFile>':
5   dyld                                0x195539d54 start + 7184
4   data_processer                      0x1021c0b90 main + 12
3   data_processer                      0x1021c0b1c startDataIngestion() + 208
2   data_processer                      0x1021c0944 processLargeFile(...) + 184
1   libstdc++.6.dylib                   0x1026c85fc operator new(unsigned long) + 28
0   libsystem_malloc.dylib              0x195725f00 _malloc_zone_malloc + 152

20000 (469K) << TOTAL >>
```

**Result:** Massive memory leak confirmed:

- 10,000 `DataRecord` objects never freed
- 10,000 `data_buffer` arrays never freed
- Total: 20,000 leaks for 480KB

## 2.4 Project Organization

### Directory Structure

```
hw1/
  prob_src/              # Original faulty source files
     ResourceScheduler.py
     DataProcessor.cpp
     LoggingSystem.cpp
     ProfileUpdater.cpp
     MatrixProcessor.cpp
  tests/                 # Test implementations
     test_deadlock.py
     test_buffer_overflow.cpp
     test_const_violation.cpp
  evidence/              # Test output evidence (preserved)
     deadlock_output.txt
     leaks_output.txt
     race_output.txt
     overflow_output.txt
     const_violation_output.txt
  verifier.cpp           # Main verification framework
  prob.js                # JavaScript homework (findLast, etc.)
  prob_test.js           # Node.js tests for JavaScript homework
  data.txt               # Input data for DataProcessor
```

| Test | Status | Evidence | Key Finding |
|---|---|---|---|
| Deadlock | PASS | Program timeout (5s) | Both threads blocked waiting for each other's locks |
| Memory Leak | PASS | 20000 leaks detected | 480KB leaked across 10,000 records |
| Race Condition | PASS | 98320/100000 processed | Lost 1680 updates due to concurrent access |
| Buffer Overflow | PASS | Memory corruption | Adjacent struct fields overwritten |
| Const Violation | PASS | SEGFAULT (exit 138) | Crash writing to read-only memory |

Table 1: Verification Test Results

## 2.5 Verification Results Summary

## 2.6 Technical Insights

### Buffer Overflow Details

The overflow works because of struct memory layout:

```
Memory Layout of UserProfile:
[username: 1 byte][padding: 3 bytes][user_id: 4 bytes][profile_status: 1 byte]
[padding: 3 bytes][is_active: 1 byte][padding: 3 bytes][last_login_year: 4 bytes]

When strcpy writes 52 bytes into 1-byte buffer:
- Overwrites padding
- Overwrites user_id
- Overwrites profile_status
- Overwrites is_active
- Overwrites last_login_year
- May crash or corrupt further memory
```

### Const Violation Key Learning

**Why local const didn't crash but global const did:**

- **Stack variables:** Even if declared const, compiler may place on writable stack
- **Global/static const:** Placed in `.rodata` section (read-only data segment)
- **Attempting to write to .rodata:** Hardware memory protection triggers SEGFAULT

### Race Condition Math

With 10 threads doing 10,000 increments each:

- Expected: 100,000
- Typical result: 98,000-99,000
- Lost updates: 1,000-2,000 (1-2% race window)

The race window exists during the read-modify-write cycle:

```
LOAD  total_logs_processed -> register  # Thread A reads
INC   register                          # Thread A increments
LOAD  total_logs_processed -> register  # Thread B reads (same value!)
STORE register -> total_logs_processed  # Thread A writes
STORE register -> total_logs_processed  # Thread B writes (overwrites A!)
```

### 2.7 Compilation and Execution

#### Compile Verifier

```
g++-15 -std=c++11 verifier.cpp -o verifier
```

#### Run All Tests

```
./verifier
```

#### Individual Test Compilation

```
# Deadlock test
python3 tests/test_deadlock.py

# Buffer overflow test
g++-15 -std=c++11 -fpermissive tests/test_buffer_overflow.cpp -o test_buffer_overflow
./test_buffer_overflow

# Const violation test
g++-15 -std=c++11 -fpermissive tests/test_const_violation.cpp -o test_const_violation
./test_const_violation

# Race condition test
g++-15 -std=c++11 -pthread prob_src/LoggingSystem.cpp -o logging_system
./logging_system

# Memory leak test
g++-15 -std=c++11 prob_src/DataProcessor.cpp -o data_processer
leaks -atExit -- ./data_processer
```

## 3 Conclusion

All five faulty programs have been successfully verified with comprehensive test cases. The automated verification framework now:

**Final Verification Summary:**

```
Passed:  5
Failed:  0
Skipped: 0 (with gtimeout installed)
```

All evidence files preserved in `evidence/` directory for review and grading.